Владислав ПИРОГОВ



АССЕМБЛЕР и ДИЗАССЕМБЛИРОВАНИЕ



ИНСТРУМЕНТАРИЙ ИССЛЕДОВАНИЯ ИСПОЛНЯЕМОГО КОДА

ФОРМАТ КОМАНД МИКРОПРОЦЕССОРА INTEL

ПЕРЕПОЛНЕНИЕ БУФЕРА

ПРИМЕРЫ ИССЛЕДОВАНИЯ ИСПОЛНЯЕМОГО КОДА

ДИЗАССЕМБЛИРОВАНИЕ И ОТЛАДКА В IDA Pro И SoftICE



Владислав Пирогов

Ассемблер и дизассемблирование

Санкт-Петербург
«БХВ-Петербург»
2006

УДК 681.3.068+800.92Ассемблер
ББК 32.973.26-018.1
П33

Пирогов В. Ю.

П33 Ассемблер и дизассемблирование. — СПб.: БХВ-Петербург, 2006. — 464 с.: ил.

ISBN 5-94157-677-3

Рассмотрены вопросы исследования кода Windows-приложений. Подробно описаны формат исполняемых модулей и структура инструкций микропроцессора Intel. Дан полный обзор инструментария по исследованию исполняемого кода: отладчики, дизассемблеры, редакторы ресурсов, HEX-редакторы и др. Большое внимание уделено работе с популярными программами по дизассемблированию и отладке SoftICE и IDA Pro. Приведены примеры исследования исполняемого кода и описаны основные принципы подобного исследования: идентификация программных структур, поиск данных и др. Прилагаемый компакт-диск содержит тексты всех листингов, описанных в книге, а также учебные программы.

Для программистов

УДК 681.3.068+800.92Ассемблер
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Наталия Першакова*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 20.02.06.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 37,41.
Тираж 3000 экз. Заказ № 119
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию № 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 5-94157-677-3

Оглавление

Введение

Конечно, скажете вы, дорогие читатели, исправлять чужую программу нехорошо и даже противозаконно. Когда-то в эпоху незабвенной MS-DOS автор этих строк написал небольшой резидентный драйвер для принтера. В те далекие времена часто возникала проблема русификации либо перекодировки принтеров. И вот спустя год я обнаружил свой драйвер в одном из учреждений, который был установлен неким господином X. Но драйвер был не только установлен, была изменена и подпись, так что получилось, что драйвер был написан этим господином. Неприятный осадок у меня остался до сих пор, хотя на этого человека я уже давно не сержусь. Так что мне понятны чувства авторов, программы которых подверглись незаконному исследованию и изменению.

Но закрывать глаза на то, что объективно существует, мы также не можем. Мы должны знать оружие врага, чтобы эффективно строить защиту своих собственных программ. Атаки взломщиков и компьютерные вирусы в действительности играют и положительную роль — они заставляют разработчиков программного обеспечения более внимательно относиться к своим продуктам. В небольших дозах атаки на софт и компьютерные вирусы — это стимуляторы иммунной системы ПО, хотя в большом количестве это может привести к "заболеванию" и даже "смерти". В книге даны примеры исследования и исправления исполняемого кода, но все они приводятся только в качестве учебного материала.

Есть и другие причины исследовать исполняемый код. Понимание исполняемого кода, т. е. того, как те или иные структуры языка высокого уровня преобразуются в ассемблерные команды, может помочь нам в написании более эффективных и оптимизированных программ. Часто для понимания ошибки, которая возникает при работе программы, требуется низкоуровневая отладка. Наконец, как мне кажется, любому профессиональному программисту просто интересно понять, как работают его программы, во что превращается текст программы, скажем, на C++ или Delphi, после того как над ним поработает транслятор. Так что лишний раз повторюсь, что все примеры, которые приводятся в данной книге, направлены на положительные цели и ни в коем случае не на противоправные действия.

Я не планировал эту книгу как учебное пособие, хотя время появления таковых в данной области уже, наверное, пришло. Это скорее попытка систематизировать материал, который уже давно копился у меня и до сих пор не находил воплощения. Возможно, в будущем есть смысл сделать из данной книги учебник по исследованию исполняемого кода, чем я с удовольствием займусь.

Значительное место в книге уделено таким мощным средствам исследования исполняемого кода, как дизассемблер IDA Pro и отладчик SoftICE. Я очень надеюсь, что читатель возьмет на вооружение эти по истине безграничные по своим возможностям инструменты.

В книге довольно много справочного материала. Возможно, это признаки типичной программисткой болезни, которая проявляется в попытках написания самодостаточной универсальной программы, что чаще всего лишь несбыточная мечта. Мне все же кажется, что не хватает книг, которые бы не вынуждали читателя после прочтения каждых десяти страниц усиленно рыться в других книгах и Интернете.

При написании книги я ориентировался на операционные системы семейства Windows NT/2000/XP/Server 2003. Хотя многое, что содержится в данной книге, будет, несомненно, справедливо и для операционных систем Windows 9*x*, но я не проверял материал на этих системах.

Большинство примеров, которые я разбираю в данной книге, относятся к языку C++ и компилятору Visual C++. В меньшей степени я рассматриваю компилятор Borland C++ 5.0, еще в меньшей степени касаюсь языка и компилятора Delphi. "Почему такое ограничение?" — спросите вы. Просто я выбрал за основу классический язык программирования и самый мощный и популярный компилятор.

К книге прилагается компакт-диск с листингами и графическими схемами, используемыми в книге.

Для кого эта книга

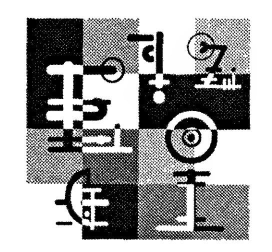
Прежде всего, книга не предназначена для читателей, не знакомых с программированием. Если вы, дорогие друзья, программируете на каком-либо языке высокого уровня, но не знаете языка ассемблера, то вам время от времени все же придется заглядывать и в какую-нибудь книжку, где толково говорится об этом языке. Вполне может подойти и моя книга "Ассемблер на примерах" [2]. Я привожу также много примеров и на языке C++, которые, я уверен, не вызовут затруднений у программирующих читателей.

Надеюсь, что книга будет полезна всем, кто интересуется внутренним устройством программ, механизмом их работы, тем как операторы языка высокого уровня превращаются в машинные команды, в общем, всем компьютерщикам, в ком бьется исследовательская жилка и кто интересуется секретами программирования.

Благодарности

Благодарю Шишигина Игоря, который предложил мне написать данную книгу. Я получил массу удовольствия, работая над ней, и теперь очень надеюсь, что и читатель найдет данную книгу полезной и приятной во всех отношениях.

Глава 1



Введение в дизассемблирование

Ассемблер и *дизассемблер* — две стороны одной медали. Ассемблер превращает текст программы на языке ассемблера в двоичный код, дизассемблер переводит модуль в двоичном коде в последовательность ассемблерных команд. Для анализа дизассемблированного кода, таким образом, крайне важно знать машинные команды, их двоичный формат и ассемблерное представление. Очень важно понимать структуру представления данных в памяти компьютера, а также знать структуру программ, написанных для операционной системы Windows. Все это мы будем обсуждать в данной главе.

1.1. Представление информации в памяти компьютера

Цель данного раздела — посмотреть, как числовые данные размещаются в памяти компьютера.

1.1.1. Заглянем в память

Обратимся к простой программе на языке С (листинг 1.1)[1]. Программа призвана вывести содержимое области памяти, начиная с блока, где хранится значение переменной k. Такая выведенная на какое-либо устройство область памяти называется *дампом* (от англ. *dump*). В листинге 1.1 представлена

---

[1] Здесь и далее все программы на языке С будут компилироваться с помощью Visual C++ (из Visual Studio .NET 2003), лучшим, на мой взгляд, на сегодняшний день транслятором языка С++. Особые случаи мы обговорим отдельно.

программа, которая выводит на текстовый экран область памяти, где хранятся переменные.

Листинг 1.1

```
#include <stdio.h>
#include <windows.h>
int k=0x1667;
BYTE  *b=(BYTE*)&k;
void main()
{
  int j=0;
  printf("\n%p   ",b);
  for(int i=0; i<400; i++)
  {
    printf("%02x  ",*(b++));
    if(++j==16&&i<398){
      printf("\n");
      j=0;
      printf("%p   ",b);
    };
  };
  printf("\n");
};
```

Воспользуемся менеджером Far и запустим откомпилированную нами программу. На экран консоли будет выведена таблица шестнадцатеричных чисел (рис. 1.1).

Судя по рисунку, в памяти, кроме значения переменной `k`, равного 0x1667 (младший байт слова имеет меньший адрес), находятся и другие данные. Что это за данные? Как разобраться в таблицах шестнадцатеричных чисел? Мы начнем наше рассмотрение с элементарных, на взгляд, я думаю, многих подготовленных читателей, вопросов, связанных с представлением чисел в памяти компьютера. Те, кто хорошо владеет этим материалом, спокойно могут пропустить *разд. 1.1.2* и *1.1.3*.

```
E:\programs\c++\memory\Release>memory.exe

00408040   67  16  00  00  44  80  40  00  14  1e  40  00  01  00  00  00
00408050   94  61  40  00  84  61  40  00  80  8c  40  00  00  00  00  00
00408060   80  8c  40  00  01  01  00  00  00  00  00  00  00  00  00  00
00408070   00  10  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00408080   00  00  00  00  02  00  00  00  01  00  00  00  00  00  00  00
00408090   00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
004080A0   00  00  00  00  02  00  00  00  02  00  00  00  00  00  00  00
004080B0   00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
004080C0   00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
004080D0   00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
004080E0   00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
004080F0   00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00408100   00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00408110   00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00408120   00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00408130   00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00408140   00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00408150   00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00408160   00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00408170   00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00408180   00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
00408190   00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
004081A0   00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
004081B0   00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
004081C0   00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00

E:\programs\c++\memory\Release>
```

Рис. 1.1. Дамп, выводимый программой из листинга 1.1

1.1.2. Системы счисления

Десятичная система счисления

Десятичная система счисления знакома нам с детства. Она естественна для нас и освещена вековыми традициями. Двоичная система счисления не привычна для нас, но естественна для компьютера. Память его состоит из элементов, которые могут находиться в двух состояниях. Поэтому логично, что одно состояние принято обозначать как 0, а другое как 1. В результате вся информация в памяти записывается в виде двоичных чисел, т. е. последовательности нулей и единиц. Кроме этого память также делят на блоки по восемь элементов, которые называют *ячейками памяти* или *байтами*. Один разряд в двоичной записи числа называют также *битом*. Таким образом, каждая ячейка памяти будет состоять из восьми двоичных разрядов или восьми битов.

Вспомним, что десятичные числа — это числа по основанию 10, т. е. любое десятичное число может быть представлено в виде суммы по степеням 10, где коэффициентами служат разряды числа. Вот так:

$$4567 = 4\times10^3 + 5\times10^2 + 6\times10^1 + 7\times10^0.$$

Другими словами, каждый разряд дает вклад в зависимости от позиции, которую он занимает. Позиция эта определяется номером справа налево, начиная с нуля. Такие системы счисления называют еще *позиционными*.

Двоичная система счисления

Двоичная система счисления также является позиционной. Любое двоичное число, таким образом, может быть представлено в виде суммы степеней числа 2:

$$11101001 = 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0.$$

Данный способ записи двоичного числа является, по сути, и способом перевода его в другую систему счисления. В частности, выполнив действия в десятичной системе счисления, мы получим число 233.

Перевести десятичное число в двоичное представление несколько сложнее. Это делается по следующей схеме:

1. Делим число на 2.
2. Если результат больше единицы, то возвращаемся к пункту 1.
3. Двоичное число составляется из последнего результата деления (старший бит), а также всех остатков от деления.

Рассмотрим перевод в двоичную систему счисления числа 350 (рис. 1.2).

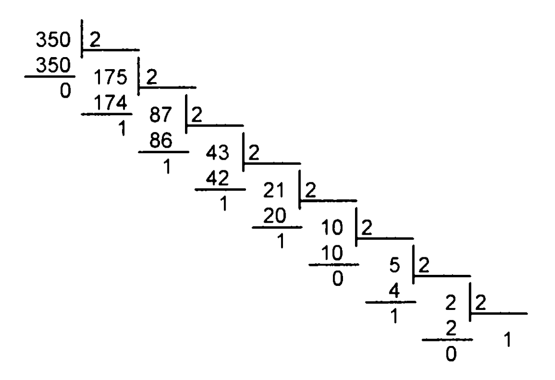


Рис. 1.2. Перевод в двоичную систему счисления числа 350

В результате представленных выше вычислений можно видеть, что двоичным представлением числа 350 будет 101011110.

Для того чтобы в программах на языке ассемблера различать числа в разных системах счисления, в конце записи числа используется однобуквенное окончание: для двоичной системы счисления это окончание B. Для

десятичных чисел используется окончание D, которое может быть опущено. Для шестнадцатеричных чисел окончание H. Например, 10000B, 345H, 100 и т. д.

По аналогии с десятичными дробями можно рассматривать двоичные дроби. Например, двоичное число 1001,1101 можно представить как:

$$1\times 2^3+0\times 2^2+0\times 2^1+1\times 2^0+1\times (1/2)^1+1\times (1/2)^2+0\times (1/2)^3+1\times (1/2)^4.$$

Очевидно, что перевод двоичной дроби в десятичную систему счисления производится также просто выполнением действий. Переведем, например, число 1001,1101 в десятичный формат. Для этого выполним все указанные в представлении двоичного числа действия. В результате получаем десятичную дробь 9,8125.

Перевод десятичной дроби в двоичную систему производится также достаточно просто. Целая и дробная части дроби переводятся отдельно. Как переводится целая часть, мы уже знаем. Перевод дробной части осуществляется с помощью следующих шагов:

1. Нужно умножить дробную часть на 2 (основание системы).
2. В полученном числе необходимо выделить целую часть (это будет либо 0, либо 1) — это и будет первый после запятой разряд в двоичной системе счисления.
3. Если дробная часть получившегося числа отлична от нуля, то следует перейти к пункту 1, в противном случае закончить вычисления. Можно установить точность вычисления, т. е. количество полученных после запятой разрядов и прекратить вычисления по достижению этой точности.

$0{,}406\times 2$	$(1/2)^1\times 0$
$0{,}812\times 2$	$(1/2)^2\times 1$
$0{,}624\times 2$	$(1/2)^3\times 1$
$0{,}248\times 2$	$(1/2)^4\times 0$
$0{,}496\times 2$	$(1/2)^5\times 0$
$0{,}992\times 2$	$(1/2)^6\times 1$
$0{,}984\times 2$	$(1/2)^7\times 1$
$0{,}968\times 2$	$(1/2)^8\times 1$
$0{,}936\times 2$	$(1/2)^9\times 1$

Рис. 1.3. Перевод дробной части числа 105,406 в двоичную систему счисления

Рассмотрим конкретный пример преобразования десятичного числа 105,406 в двоичное представление. Как перевести целую часть числа, мы уже знаем. Таким образом, число 105 в двоичном представлении — это 1101001. Для перевода дробной части воспользуемся описанной схемой. На рис. 1.3 представлена последовательность вычислений. Замечу, что мы оказались вынужденными остановиться на девяти знаках после запятой.

В результате наших вычислений получаем, что

$$105{,}406 \approx 1101001{,}011001111.$$

Таким образом, перевод десятичных чисел в двоичный формат, в котором они хранятся в памяти компьютера — это дополнительный фактор потери точности.

Шестнадцатеричная система счисления

Шестнадцатеричная система счисления более компактна, чем десятичная система счисления. Числа в шестнадцатеричной системе легко переводятся в двоичную систему и обратно, и, наконец, она лучше всякой другой системы соответствует архитектуре памяти компьютера. Для записи чисел в этой системе счисления используются шестнадцать символов (десять цифр и шесть букв): 0, 1, ..., 9, A, B, C, D, E, F. Способ перевода числа из десятичной системы счисления в шестнадцатеричную систему и обратно аналогичен способу, описанному в предыдущем разделе, с той лишь разницей, что здесь основанием системы счисления является 16, а не 2. Я думаю, что читатель без труда справится с этим вопросом самостоятельно.

Остановимся на переводе чисел из шестнадцатеричной системы в двоичную систему и обратно. Принцип здесь чрезвычайно прост: каждому разряду шестнадцатеричного числа соответствует четыре разряда *(тетрада)* двоичного и числа и наоборот. На рис. 1.4 представлен пример преобразования двоичного числа 10101101 в шестнадцатеричную систему счисления.

Рис. 1.4. Преобразование двоичного числа 10101101 в шестнадцатеричную систему счисления

На рис. 1.5 представлено обратное преобразование шестнадцатеричного числа `14A` в двоичное число.

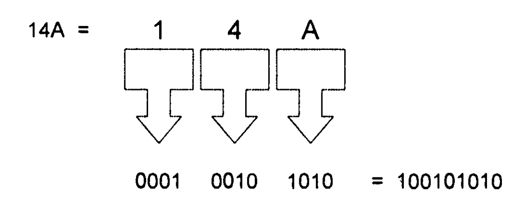


Рис. 1.5. Преобразование шестнадцатеричного числа `14A` в двоичное число

Итак, шестнадцатеричная система счисления хорошо укладывается в архитектуру памяти компьютера. Действительно, память компьютера разбивается на ячейки по восемь битов. Но восемь битов как раз соответствуют двум разрядам в шестнадцатеричной записи числа. Например, очевидно, что число `1345H` будет занимать две ячейки памяти, причем в младшей ячейке (так принято), т. е. ячейке с меньшим адресом, будет находиться `45H`, а в старшей — `13H`.

В случае дробных чисел перевод из шестнадцатеричной системы счисления в двоичную систему и обратно производится также просто, как это делается с целыми числами. Дробная часть, как и целая, переводится по принципу: один разряд в шестнадцатеричной системе соответствует четырем разрядам двоичной системы. Рассмотрим двоичное дробное число 101,10001 и переведем его в шестнадцатеричную системы счисления. Имеем 101 → 0101 → 5. Далее дробная часть 10001 → 10001000 → 88 (заметьте, что четверки разрядов в дробной части отсчитываются слева направо). В результате получаем, что число 101,10001 в двоичной системе счисления равно числу 5,88 в шестнадцатеричной системе счисления. Как и в случае целой части, перевод дробной части сводится к выделению четверок и дополнении нулями неполных четверок (но только справа).

1.1.3. Представление чисел в компьютере

Беззнаковые целые числа

Принцип представления беззнаковых целых чисел в компьютере достаточно тривиален:

- число должно быть переведено в двоичную систему счисления;
- должен быть определен объем памяти для этого числа.

Как мы уже говорили, это удобно делать, когда число представлено в шестнадцатеричной системе счисления, и тогда станет ясно, сколько ячеек памяти необходимо для хранения этого числа. Принято выделять: одинарные ячейки памяти *(байты)*, двойные ячейки *(слова)*, четвертные ячейки *(четыре байта)*. В языке ассемблера имеются специальные директивы для резервирования памяти для числовых констант и переменных:

```
Имя1 DB значение 1 ; резервируем один байт
Имя2 DW значение 2 ; резервируем два байта
Имя3 DD значение 3 ; резервируем четыре байта
Имя4 DQ значение 4 ; резервируем восемь байтов
Имя5 DT значение 4 ; резервируем десять байтов
```

Если речь идет о переменной, а чаще так и бывает, необходимо определить диапазон, в котором будет меняться значение переменной и, исходя из этого, резервировать для нее память. Поскольку современные процессоры Intel ориентированы на операции с 32-битными числами, то оптимальнее все же ориентироваться на переменные такой же размерности.

Рассмотрим фрагмент программы на языке С.

```
BYTE e=0xab;
WORD c=0x1234;
DWORD b=0x34567890;
__int64 a=0x6178569812324572;
```

В данном фрагменте задано четыре переменных: однобайтовая e, двухбайтовая c, четырехбайтовая b[2], восьмибайтовая a. С помощью программы, которую я продемонстрировал в листинге 1.1, выведем область памяти, где хранятся эти переменные. Вот полученная последовательность байтов:

```
ab 00 00 00 34 12 00 00 90 78 56 34 00 00 00 00 72 45 32 12 98 56 78 61
```

Внимательно посмотрев на данную цепочку байтов, вы без труда обнаружите все наши переменные. Что важного можно почерпнуть из данной последовательности?

- Как вы помните, в листинге 1.1 мы выводили содержимое памяти в сторону старших адресов. Таким образом, мы видим, что младшие байты чисел (переменных) в слове занимают в памяти младшие адреса. В свою очередь младшие слова в удвоенном слове — младший адрес. И, наконец, если рассматривать 64-битную переменную, то в ней младшее удвоенное слово должно занимать младший адрес. Это очень важный момент именно для анализа двоичного кода. В дальнейшем по

[2] BYTE — это просто unsigned char, WORD — unsigned short int, DWORD — unsigned int. Определение этих типов есть, например, в заголовочном файле windows.h.

одному виду области памяти вы сможете во многих случаях сразу идентифицировать переменные.

- Как видно, на все переменные затрачивается объем памяти, кратный четырехбайтовой величине. После каждой инициализированной переменной компилятор вставляет директиву выравнивания по 32-битной границе (`Align 4`). Впрочем, все совсем не так просто, и при другом порядке следования переменных выравнивание может быть иным. К данному вопросу мы вернемся в *разд. 3.1.1*.

Примеры

Итак. 16-битное число `A890H` в памяти будет храниться как последовательность байтов `90 A8`. 32-битное число `67896512H` — как последовательность `12 65 89 67`. И, наконец, 64-битное число `F5C68990D1327650H` — как `50 76 32 D1 90 89 C6 F5`.

Числа со знаком

Поскольку в памяти нет ничего, кроме двоичных разрядов, то вполне логично было бы выделить для знака числа отдельный бит. Например, имея одну ячейку, мы могли бы получить диапазон чисел от −127 до 127 (11111111—01111111). Подход был бы не так уж и плох. Вот только пришлось бы вводить отдельно сложение для знаковых и беззнаковых чисел. Существует и другой, альтернативный способ введения знаковых чисел. Алгоритм построения заключается в том, что мы объявляем некоторое число заведомо положительным и далее ищем для него противоположное по знаку исходя из очевидно тождества: $a + (-a) = 0$.

На множестве однобайтовых чисел за единицу естественно взять двоичное число 00000001. Решая уравнение $00000001 + x = 00000000$, мы приходим к неожиданному на первый взгляд результату $x = 11111111$; другими словами, за −1 мы должны принять число 11111111 (255 в десятичном эквиваленте и FF в шестнадцатеричном). Попробуем развить нашу теорию. Очевидно, что $-1 - (1) = -2$, т. е., по логике вещей, за −2 мы должны принять число 11111110. Но с другой стороны число 00000010 вроде бы должно представлять +2. Посмотрите 11111110 + 00000010 = 00000000, т. е. выполняется очевидное тождество $+2 + (-2) = 0$. Итак, мы на верном пути и процесс можно продолжить (табл. 1.1).

Таблица 1.1. *Знаковые однобайтовые числа*

Положительные числа	Двоичное представление	Отрицательные числа	Двоичные представления
+0	00000000	−0	00000000
+1	00000001	−1	11111111

Таблица 1.1 (окончание)

Положительные числа	Двоичное представление	Отрицательные числа	Двоичные представления
+2	00000010	–2	11111110
+3	00000011	–3	11111101
+4	00000100	–4	11111100
+5	00000101	–5	11111011
...	...	...	...
+120	01111000	–120	10001000
+121	01111001	–121	10000111
+122	01111010	–122	10000110
+123	01111011	–123	10000101
+124	01111100	–124	10000100
+125	01111101	–125	10000011
+126	01111110	–126	10000010
+127	01111111	–127	10000001
+128	Не существует в пределах 1 байта	–128	10000000

Внимательно посмотрите на таблицу. Что же у нас получилось? Знаковые числа оказываются в промежутке –128 до 127.

Таким образом, однобайтовое число можно интерпретировать и как число со знаком, и как беззнаковое число. Тогда, например, 11111111 в первом случае будет считаться –1, а во втором случае 255. Все зависит от нашей интерпретации. Еще интереснее операции сложения и вычитания. Эти операции будут выполняться по одной и той же схеме и для знаковых, и для беззнаковых чисел. По этой причине и для операции сложения, и для операции вычитания у процессора имеется всего по одной команде: ADD и SUB. Разумеется, при выполнении действия может возникнуть переполнение или перенос в несуществующий разряд[3], но это отдельный разговор, и решить проблему можно, зарезервировав еще одну ячейку памяти. Все наши рассуждения легко переносятся на двух- и четырехбайтовые числа. Так, максимальное двухбайтовое беззнаковое число будет 65 535, а знаковые чис-

[3] Легко показать, что возможность одновременного представления и знаковых, и беззнаковых чисел заложена в том, что мы ограничиваем размер числа одним или несколькими байтами.

ла окажутся в промежутке от −32 768 до 32 767. Еще один интересный момент касается старшего бита. Как мы видим, по нему можно определить знак. Но в данной схеме бит совсем не изолирован и участвует в формировании значения числа вместе с остальными битами.

Уметь хорошо ориентироваться в знаковых и беззнаковых числах очень важно для исследователя программного кода. Встретив, скажем, команду

```
cmp eax, 0FFFFFFFEh
```

следует иметь в виду, что в действительности это, возможно, команда

```
cmp eax, -2
```

Рассмотрим последовательность переменных:

```
signed char e=-2;
short int c=-3;
int b=-4;
__int64 a=-5;
```

Как видим, это все знаковые переменные с отрицательным значением. При выводе фрагмента памяти, содержащего данные переменные, получим следующую последовательность байтов:

```
fe 00 00 00 fd ff 00 00 fc ff ff ff 00 00 00 00 fb ff ff ff ff ff ff ff
```

Итак, значение однобайтовой переменной `-2` в памяти компьютера представлено байтом `FEH`, значение двухбайтовой переменной `-3` представлено последовательностью `FFFDH`, значение четырехбайтовой переменной `-4` — последовательностью `FFFFFFFCH`, и, наконец, отрицательная восьмибайтовая переменная со значением `-5` представлена байтами `FFFFFFFFFFFFFFFBH`. Напоминаю, что при представлении восьмибайтового числа младшие четыре байта должны находиться по адресу, меньшему, чем старшие.

Вещественные числа

Для того чтобы использовать вещественные числа в командах процессора Intel (командах арифметического сопроцессора[4]), они должны быть представлены в памяти компьютера в *нормализованном виде*. В общем случае нормализованный вид числа выглядит так:

$$A = (ZN) \times M \times N^q.$$

Здесь ZN — знак числа; M — *мантисса* числа, обычно удовлетворяет условию $M < 1$; N — основание системы счисления; q — показатель, в общем случае может быть и положительным, и отрицательным числом. Числа, пред-

[4] Начиная с процессора Intel 486, арифметический сопроцессор является его частью.

ставленные таким образом, называют еще *числами с плавающей точкой* (или *числами с плавающей запятой*).

Рассмотрим конкретный пример. Попытаемся представить в нормализованном виде число 5,75. Переведем вначале это число в двоичную систему счисления. В данном случае это делается достаточно легко. Действительно, 5 — это 1001, а 0,75 — это (1/2) + (1/4). Другими словами 5,75 = 1001,11B. Пишем далее 1001,11B = 1,00111 × 2^3. Таким образом, мы имеем следующие компоненты нормализованного числа: $ZN = +1$, $M = 1{,}00111$, $N = 2$, $q = 3$.

Заметим, что первая цифра мантиссы в таком представлении всегда равна 1, а, следовательно, ее можно и вообще не хранить, и в формате Intel так и поступают. Кроме этого нужно иметь в виду, что показатель q в действительности (для процессора Intel) хранится в памяти в виде суммы с некоторым числом, так чтобы всегда быть положительным. Процессор Intel может работать с тремя типами вещественных чисел:

- *короткое вещественное число*. Всего для хранения отводиться 32 бита. Биты 0—22 резервируются для мантиссы. Биты 23—30 предназначены для хранения показателя q, сложенного с числом 127. Последний, 31-й бит, предназначен для хранения знака числа (1 — знак отрицательный, 0 — положительный);
- *длинное вещественное число*. Для хранения такого числа отводится 64 бита. Биты 0—51 нужны для хранения мантиссы. Биты 52—62 предназначены для хранения числа q, сложенного с числом 1024. Последний, 63-й бит, определяет знак числа (1 — знак отрицательный, 0 — положительный);
- *расширенное вещественное число*. Для хранения числа отводится 80 битов. Биты 0—63 — мантисса числа. Биты 64—78 — показатель q, сложенный с числом 16 383. 79-й, последний бит отводится для знака числа (1 — знак отрицательный, 0 — положительный).

Очевидно, пришла пора разобрать конкретный пример представления в памяти вещественного числа. Итак, пусть в программе на языке С имеем объявление переменной:

```
float a=-80.5
```

Тип `float` — это короткое вещественное число, т. е. в памяти оно, согласно вышезаписанному, будет занимать 32 бита. Попытаемся теперь нашим обычным способом заглянуть в память. Вот они, четыре байта, которые и призваны представлять наше число:

```
00 00 a1 c2
```

Чтобы легче было разбираться, представим последовательность из четырех байтов в двоичном виде:

```
00000000 00000000 10100001 11000010
```

Или более понятным способом, начиная со старшего байта для выделения мантиссы, показателя и знака:

```
11000010 10100001 00000000 00000000
```

Выделим мантиссу. На нее отводится 23 бита. Имеем, таким образом, двоичное число 0100001. Учтите, что биты мантиссы отсчитываются, начиная со старшего (в данном случае 22-го) бита, а оставшиеся нули естественно отбрасываются, поскольку вся мантисса располагается справа от запятой. Правда, это еще не совсем мантисса. Как ранее было сказано, единица перед запятой в представлении отбрасывается. Так что мы должны восстановить ее. Поэтому мантиссой будет число 1,0100001B. Знак всего числа, как мы видим, определяется единицей, следовательно, задает отрицательное число. А вот показатель нам следует получить из двоичного числа 10000101B. В десятичном представлении это число 133. Вычитая число 127 (для короткого вещественного числа), получим 6. Следовательно, для того чтобы получить из мантиссы истинное дробное число, нужно сместить в ней точку на шесть разрядов вправо. Окончательно получаем 1010000,1B. В шестнадцатеричной системе счисления это просто 50,8H, а в десятичной получаем как раз 80,5.

В качестве тренировки я бы вам предложил следующую цепочку байтов:

```
00 80 fb 42
```

Попытайтесь доказать, что это есть не что иное, как представление числа 125,75.

Из сказанного в данном разделе можно сделать вывод, что при использовании в программе вещественных чисел они могут стать приближенными еще до того, как с ними были произведены какие-либо действия. Это вызвано тем, что для записи чисел в память их нормализуют.

Двоично-десятичные числа

Двоично-десятичные числа (Binary-Coded Decimal, BCD) — особый способ представления десятичных чисел в памяти компьютера, когда каждому десятичному разряду ставится в соответствие фиксированное число двоичных разрядов. Процессор Intel поддерживает два вида таких чисел: упакованный и неупакованный.

- Каждый разряд *упакованного числа* кодируется четырьмя битами. При этом в старших четырех битах содержится старшая цифра. Таким образом, в байте может содержаться число в промежутке от 0 до 99. Например, число 56 будет представлено двоичным число 01010110B.
- Каждый разряд *неупакованного числа* кодируется одним байтом. Значимыми являются четыре младших байта, а четыре старших байта должны

содержать нули. Таким образом, в одном байте можно представить число от 0 до 9.

BCD-числа не так часто используются в программировании, поэтому мы не будем больше обращаться к этому вопросу.

1.2. Обзор команд и регистров микропроцессора Intel Pentium

Данный раздел целиком посвящен обзору команд микропроцессора Intel Pentium. В книге по исследованию кода исполняемых модулей, на мой взгляд, такой материал весьма полезен в качестве справочника, который всегда под рукой.

1.2.1. Регистры микропроцессора Pentium

Микропроцессор Pentium включает в себя регистры общего назначения, регистр флагов, сегментные регистры, управляющие регистры, системные адресные регистры, а также отладочные регистры. Особо следует отметить регистр `EIP`, который называют указателем команд. В нем всегда содержится адрес исполняемой команды относительно начала сегмента. К данному регистру нет прямого доступа, но косвенно многие команды изменяют его содержимое, например, команды передачи управления.

Регистры общего назначения

Перечислю их:

```
EAX = (16+AX=(AH+AL))
EBX = (16+BX=(BH+BL))
ECX = (16+CX=(CH+CL))
EDX = (16+DX=(DH+DL))
ESI = (16+SI)
EDI = (16+DI)
EBP = (16+BP)
ESP = (16+SP)
```

Регистры `EAX`, `EBX`, `EDX`, `ECX` называют *рабочими регистрами*. Обратите внимание, что эти регистры имеют подрегистры. Например, первые 16 битов регистра `EAX` обозначаются как `AX`. В свою очередь младший байт `AX` обозначается как `AL`, а старший байт как `AH`. Регистры `EDI`, `ESI` — *индексные регистры*, играют особую роль в строковых операциях. Регистр `EBP` обычно используется для адресации в стеке параметров и локальных переменных.

Регистр ESP — указатель стека, автоматически модифицируется командами PUSH, POP, RET, CALL, но явно используется реже. Регистры ESI, EDI, ESP, EBP также имеют подрегистры. Например, первые 16 битов регистра EDI обозначаются как DI.

Регистр флагов

Содержит 32 бита. Далее приведены используемые значения битов этого регистра.

- 0-й бит, флаг переноса (CF), устанавливается в 1, если был перенос из старшего бита при операции сложения или заем бита при операции вычитания.
- 1-й бит, 1.
- 2-й бит, флаг четности (PF). Устанавливается в 1, если младший байт результата содержит четное число единиц, и в 0 — в противном случае.
- 3-й бит, 0.
- 4-й бит, флаг вспомогательного переноса (AF). Устанавливается в 1, если произошел перенос (или заем) из третьего бита в четвертый.
- 5-й бит, 0.
- 6-й бит, флаг нуля (ZF). Устанавливается в 1, если результат операции — ноль, и в 0 — в противном случае.
- 7-й бит, флаг знака (SF). Равен старшему биту результата последней команды.
- 8-й бит, флаг ловушки (TF). Установка в 1 этого флага приводит к тому, что после каждой команды вызывается INT 3. Используется отладчиками в реальном режиме.
- 9-й бит, флаг прерываний (IF). Сброс этого флага в 0 приводит к тому, что микропроцессор перестает воспринимать прерывания от внешних устройств.
- 10-й бит, флаг направления (DF). Данный флаг учитывается в строковых операциях. Если флаг равен 1, то в строковых операциях адрес автоматически уменьшается.
- 11-й бит, флаг переполнения (OF). Устанавливается в единицу, если результат операции над числом со знаком вышел за допустимые пределы.
- 12-й, 13-й биты, уровень привилегий ввода/вывода (IOPL). Определяют, какой привилегией должен обладать код, чтобы ему было разрешено выполнить команды ввода/вывода, а также другие привилегированные команды.

- 14-бит, флаг вложенной задачи (NT).
- 15-й бит, 0.
- 16-й бит, флаг возобновления (RF). Используется совместно с регистрами точек отладочного останова.
- 17-й бит, в защищенном режиме включает режим виртуального режима 8086 (VM).
- 18-й бит, флаг контроля выравнивания (AC). При равенстве этого флага 1 и при обращении к невыровненному операнду вызывает исключение 17.
- 19-й бит, виртуальная версия флага IF (VIF). Работает в защищенном режиме.
- 20-й бит, виртуальный запрос прерывания (VIP).
- 21-й бит, флаг доступности команды идентификации.
- 22—31-й биты, должны содержать 0.

Сегментные регистры

CS — сегмент кода, DS — сегмент данных, SS — сегмент стека, ES, GS, FS — дополнительные регистры. Все сегментные регистры 16-битные. Назначение сегментных регистров — участвовать в формировании адреса памяти либо напрямую, либо посредством селекторов, которые указывают на некоторую структуру (в дескрипторной таблице), определяющей сегмент, где находится формируемый адрес.

Управляющие регистры

Перечислю их.

- Регистр CR0:
  - 0-й бит, разрешение защиты (PE). Переводит процессор в защищенный режим;
  - 1-й бит, мониторинг сопроцессора (MP). Вызывает исключение 7 по каждой команде WAIT;
  - 2-й бит, эмуляция сопроцессора (EM). Вызывает исключение 7 по каждой команде сопроцессора;
  - 3-й бит, бит переключения задач (TS). Позволяет определить, относится данный контекст сопроцессора к текущей задаче или нет. Вызывает исключение 7 при выполнении следующей команды сопроцессора;
  - 4-й бит, индикатор поддержки инструкций сопроцессора (ET);

- 5-й бит, разрешение стандартного механизма сообщений об ошибке сопроцессора (NE);
- 5—15-й биты, не используются;
- 16-й бит, разрешение защиты от записи на уровне привилегий супервизора (WP);
- 17-й бит, не используется;
- 18-й бит, разрешение контроля выравнивания (AM);
- 19—28-й биты, не используются;
- 29-й бит, запрет сквозной записи кэша и циклов аннулирования (NW);
- 30-й бит, запрет заполнения кэша (CD);
- 31-й бит, включение механизма страничной переадресации.

- Регистр CR1 пока не используется.
- Регистр CR2 хранит 32-битный линейный адрес, по которому был получен последний отказ страницы памяти.
- Регистр CR3 в старших 20 битах хранит физический базовый адрес таблицы каталога страниц. Остальные биты:
  - 3-й бит, кэширование страниц со сквозной записью (PWT).
  - 4-й бит, запрет кэширование страницы (PCD).
- Регистр CR4:
  - 0-й бит, разрешение использования виртуального флага прерываний в режиме V8086 (VME);
  - 1-й бит, разрешение использования виртуального флага прерываний в защищенном режиме (PVI);
  - 2-й бит, превращение инструкции RDTSC в привилегированную (TSD);
  - 3-й бит, разрешение точек останова по обращению к портам ввода/вывода (DE);
  - 4-й бит, включает режим адресации с 4-мегабайтными страницами (PSE);
  - 5-й бит, включает 36-битное физическое адресное пространство (PAE);
  - 6-й бит, разрешение исключения MC (MCE);
  - 7-й бит, разрешение глобальной страницы (PGE);
  - 8-й бит, разрешает выполнение команды RDPMC (PMC);
  - 9-й бит, разрешает команды быстрого сохранения/восстановления состояния сопроцессора (FSR).

Системные адресные регистры

Эти регистры используются в защищенном режиме процессора Intel, в котором, в частности, и функционирует операционная система Windows.

- GDTR — 6-байтный регистр, в котором содержится линейный адрес глобальной дескрипторной таблицы.
- IDTR — 6-байтный регистр, содержащий 32-битный линейный адрес таблицы дескрипторов обработчиков прерываний (IDT).
- LDTR — 10-байтный регистр, содержащий 16-битный селектор (индекс) для GDT и 8-байтный дескриптор.
- TR — 10-байтный регистр, содержащий 16-битный селектор для GDT и весь 8-байтный дескриптор из GDT, описывающий TSS текущей задачи. TSS — это сегмент специального формата, который содержит всю необходимую информацию о задаче, а также специальное поле, обеспечивающее связь задач между собой.

Регистры отладки

К ним относятся следующие регистры.

- DR0—DR3 — хранят 32-битные линейные адреса точек останова (контрольных точек). Механизм работы регистров таков: любой формируемый программой адрес сравнивается с адресами, хранящимися в регистрах, и если есть совпадение, то генерируется исключение отладки (INT 1).
- DR6 (равносильно DR4) — отражает состояние контрольных точек. Биты этого регистра устанавливаются в соответствии с причинами, которые вызвали исключение отладки. Вот значащие биты этого регистра:
  - бит 0, если значение этого бита равно нулю, то последнее исключение произошло по достижению контрольной точки, определенной в DR0;
  - бит 1, аналогичен биту 0, но для регистра DR1;
  - бит 2, аналогичен биту 0, но для регистра DR2;
  - бит 3, аналогичен биту 0, но для регистра DR3;
  - бит 13, служит для защиты регистров отладки;
  - бит 14, если значение бита равно 1, то исключение произошло из-за того, что флаг ловушки (бит 8 в регистре флагов) равен 1;
  - бит 15, если значение бита равно 1, то исключение вызвано переключением на задачу с установленным битом ловушки.
- DR7 (равносильно DR5) — управляет установкой контрольных точек. В данном регистре для каждого регистра отладки (DR0—DR3) имеются поля, определяющие условия, при которых следует сгенерировать прерывание. Первые четыре пары битов регистра (8 битов), по паре на каждый ре-

гистр, задают, будет соответствующий регистр определять контрольную точку для локальной задачи (первый бит пары должен быть равен 1) или на все задачи системы (второй бит пары равен 1). Биты с 16 по 31 регистра определяют тип доступа, при котором будет срабатывать прерывание (при выборке команды, записи или чтении из памяти), и размер данных:

- биты 16—17, 20—21, 24—25, 28—29 определяют тип доступа: 00 — по команде, 01 — на запись, 11 — считывание и запись, 10 — не используется;
- биты 18—19, 22—23, 26—27, 30—31 задают размер операнда: 00 — байт, 01 — два байта, 11 — четыре байта, 10 — не используется.

1.2.2. Основной набор команд

К основному набору я отношу все команды микропроцессора, кроме команд математического сопроцессора и команд MMX.

Принятые в табл. 1.2—1.18 обозначения:

- *dest*, *src* — операнд-источник и операнд-получатель;
- *m* — обозначает операнд, расположенный в памяти;
- *r* — обозначает операнд — регистр процессора;
- *r8*, *r16*, *r32* — 8-, 16-, 32-битные регистры процессора;
- mm — 64-битный регистр MMX;
- *m32* и *m64* — операнды, находящиеся в памяти и имеющие размер, соответственно, 32 и 64 бита;
- ir32 — обычные регистры процессора;
- *imm* — непосредственный операнд (константа) размером в 1 байт.

Таблица 1.2. *Команды пересылки данных*

Команда	Описание
`MOV` *dest,src*	Пересылка данных в регистр, из регистра, памяти или непосредственного операнда. Пересылка данных в память из регистра или непосредственного операнда. Например, `MOV AX,10` `MOV EBX,ESI` `MOV AL, BYTE PTR MEM` `MOV DWORD PTR MEM,10000H`
`XCHG` *r/m,r*	Обмен данными между регистрами или регистром и памятью. Команда "память—память" в микропроцессоре Intel не предусмотрена

Таблица 1.2 (продолжение)

Команда	Описание
BSWAP *r32*	Перестановка байтов из порядка "младший — старший" в порядок "старший — младший". Разряды 7—0 обмениваются с разрядами 31—24, а разряды 15—8 — с разрядами 23—16. Команда появилась в 486-м микропроцессоре
MOVSXB *r,r/m*	Пересылка байта с его расширением до слова или двойного слова с дублированием знакового бита: MOVSXB AX,BL MOVSXB EAX,byte ptr mem Команда появилась в 386-м процессоре
MOVSXW *r,r/m*	Пересылка слова с расширением до двойного слова с дублированием знакового бита: MOVSXW EAX,WORD PTR MEM Команда появилась в 386-м процессоре
MOVZXB *r,r/m*	Пересылка байта с его расширением до слова или двойного слова с дублированием нулевого бита: MOVSXB AX,BL MOVSXB EAX,byte ptr mem Команда появилась, начиная с 386-го процессора
MOVZXW *r,r/m*	Пересылка слова с расширением до двойного слова с дублированием нулевого бита: MOVZXW EAX,WORD PTR MEM Команда появилась, начиная с 386-го процессора
XLAT	Загрузить в AL байт из таблицы в сегменте данных, на начало которой указывает EBX (BX), при этом начальное значение AL играет роль смещения
LEA *r,m*	Загрузка эффективного адреса. Например: LEA EAX,MEM LEA EAX,[EBX] Данная команда обладает магическими свойствами, позволяющими эффективно выполнять арифметические действия. Например, команда LEA EAX,[EAX*8] умножает содержимое EAX на 8,

Таблица 1.2 *(продолжение)*

Команда	Описание
	`LEA EAX, [EAX][EAX*4]` умножает содержимое на 5. Команда `LEA ECX, [EAX][ESI+5]` эквивалента трем командам `MOV ECX,EAX` `ADD ECX,ESI` `ADD ECX,5` Имейте в виду, что команда LEA может умножать только на 2, 4, 8, поэтому в случае другого множителя умножение приходится сочетать со сложением
LDS *r,m*	Загрузить пару DS:*reg* из памяти. Причем вначале идет слово (или двойное слово), а в DS — последующее слово
LES *r,m*	Аналогично предыдущему, но для пары ES:*reg*
LFS *r,m*	Аналогично предыдущему, но для пары FS:*reg*
LGS *r,m*	Аналогично предыдущему, но для пары GS:*reg*
LSS *r,m*	Аналогично предыдущему, но для пары SS:*reg*
Набор команд условной установки первого бита байта SET*cc r/m*	Проверяет условие *cc*, если выполняется, то первый бит байта устанавливается в 1, в противном случае — в 0. Условия аналогичны в условных переходах (je, jc). Например, SETE AL. Команда появилась, начиная с 386-го микропроцессора. Ниже перечислены все варианты этой команды: • SETA/SETNBE — установить, если выше; • SETAE/SETNB — установить, если выше или равно; • SETB/SETNAE — установить, если ниже; • SETBE/SETNA — установить, если ниже; • SETC — установить, если перенос; • SETE/SETZ — установить, если ноль; • SETG/SETNLE — установить, если больше; • SETGE/SETNL — установить, если больше или равно; • SETL/SETNGE — установить, если меньше; • SETLE/SETNG — установить, если меньше или равно; • SETNC — установить, если нет переноса; • SETNE/SETNZ — установить, если меньше или равно;

Таблица 1.2 (окончание)

Команда	Описание
	• SETNO — установить, если нет переполнения; • SETNP/SETPO — установить, если нет паритета; • SETNS — установить, если нет знака; • SETO — установить, если есть переполнения; • SETP/SETPE — установить, если есть паритет; • SETS — установить, если есть знак
LAHF	Загрузить флаги в AH (устарела)
SAHF	Сохранить AH в регистре флагов (устарела)
Набор команд условной пересылки CMOV*X* *dest,src*	• CMOVA/CMOVNBE — переслать, если выше; • CMOVAE/CMOVNB — переслать, если выше или равно; • CMOVB/CMOVNAE — переслать, если ниже; • CMOVBE/CMOVNA — переслать, если ниже; • CMOVC — переслать, если перенос; • CMOVE/CMOVZ — переслать, если ноль; • CMOVG/CMOVNLE — переслать, если больше; • CMOVGE/CMOVNL — переслать, если больше или равно; • CMOVL/CMOVNGE — переслать, если меньше; • CMOVLE/CMOVNG — переслать, если меньше или равно; • CMOVNC — переслать, если нет переноса; • CMOVNE/CMOVNZ — переслать, если меньше или равно; • CMOVNO — переслать, если нет переполнения; • CMOVNP/CMOVPO — переслать, если нет паритета; • CMOVNS — переслать, если нет знака; • CMOVO — переслать, если есть переполнения; • CMOVP/CMOVPE — переслать, если есть паритет; • CMOVS — переслать, если есть знак

Таблица 1.3. Команды ввода/вывода

Команда	Описание
IN AL(AX,EAX),*port* IN AL(AX,EAX),DX	Ввод в аккумулятор из порта ввода/вывода. Порт адресуется непосредственно или через регистр DX
OUT *port*,AL(AX,EAX) OUT DX,AL(AX,EAX)	Вывод в порт ввода/вывода. Порт адресуется непосредственно или через регистр DX
[REP] INSB [REP] INSW [REP] INSD	Выводит данные из порта, адресуемого регистром DX, в ячейку памяти ES:[EDI/DI]. После ввода байта, слова или двойного слова производится коррекция EDI/DI на 1, 2 или 4. При наличии префикса REP-процесс продолжается, пока содержимое CX не станет равным 0
[REP] OUTSB [REP] OUTSW [REP] OUTSD	Выводит данные из ячейки памяти, определяемой регистрами DS:[ESI/SI], в выходной порт, адрес которого находится в регистре DX. После вывода байта, слова, двойного слова производится коррекция указателя ESI/SI на 1, 2, 4

Таблица 1.4. Инструкции работы со стеком

Команда	Описание
PUSH *r/m*	Поместить в стек слово или двойное слово. Поскольку при включении в стек слова нарушается выравнивание стека по границам двойных слов, рекомендуется в любом случае помещать в стек двойное слово
PUSH *const*	Поместить в стек непосредственный 32-битный операнд
PUSHA	Поместить в стек регистры EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP. Команда появилась, начиная с 386-го процессора
POP *r/m*	Извлечь из стека слово или двойное слово
POPA	Извлечь из стека данных в регистры EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP. Команда появилась, начиная с 386-го процессора
PUSHF	Поместить регистр флагов в стек
POPF	Извлечь данные в регистр флагов

Таблица 1.5. Инструкции целочисленной арифметики

Команда	Описание
ADD *dest,src*	Сложение двух операндов. Первый операнд может быть регистром или ячейкой памяти, второй — регистром, ячейкой памяти, константой. Операция невозможна, когда оба операнда являются ячейками памяти
XADD *dest,src*	Данная операция вначале производит обмен операндами, а затем выполняет операцию ADD. Команда введена, начиная с 486-го процессора
ADC *dest,src*	Сложение с учетом флага переноса — в младший бит добавляется бит (флаг) переноса
INC *r/m*	Инкремент операнда
SUB *dest,src*	Вычитание двух операндов. Остальное аналогично сложению (команда ADD)
SBB *dest,src*	Вычитание с учетом бита переноса. Из младшего бита вычитается бит (флаг) переноса
DEC *r/m*	Декремент операнда
CMP *r/m,r/m*	Вычитание без изменения операндов (сравнение)
CMPXCHG *r,m,a*	Сравнение с обменом. Воспринимает три операнда (регистр — операнд — источник, ячейка памяти — операнд — получатель, аккумулятор, т. е. AL, AX или EAX). Если значения в операнде-получателе и аккумуляторе равны, операнд-получатель заменяется операндом-источником, исходное значение операнда-получателя загружается в аккумулятор. Появилась, начиная с 486-го процессора
CMPXCHG8B *r,m,a*	Сравнение и обмен восьми байтов. Появилась, начиная с Pentium. Сравнивается число, находящееся в паре регистров EDX:EAX с восьмибайтным числом в памяти
NEG *r/m*	Изменение знака операнда
AAA	Коррекция после ASCII-сложения. Коррекция результата двоичного сложения двух неупакованных двоично-десятичных чисел. Например, AX содержит число 9H. Пара команд ADD AL, 8/AAA приводит к тому, что в AX будет содержаться 0107, т. е. ASCII-число 17
AAS	Коррекция после ASCII-вычитания. Коррекция результата двоичного вычитания двух неупакованных двоично-десятичных чисел. Например: MOV AX,205H ;загрузить ASCII-число 25 SUB AL,8 ;двоичное вычитание AAS В результате AX содержит код 107H, т. е. неупакованное двоично-десятичное число 17

Таблица 1.5 *(продолжение)*

Команда	**Описание**
AAM	Коррекция после ASCII-умножения. Для этой команды предполагается, что в регистре AX находится результат двоичного умножения двух десятичных цифр (диапазон от 0 до 81). После выполнения команды образуется двухбайтное произведение в регистре AX в ASCII-формате
AAD	Коррекция перед ASCII-делением. Предполагается, что младшая цифра находится в AL, а старшая — в AH
DAA	Коррекция после BCD-сложения[5]
DAS	Коррекция после BCD-вычитания
MUL *r/m*	Умножение AL (AX, EAX) на целое беззнаковое число. Результат, соответственно, будет содержаться в AX, DX:AX, EDX:EAX
IMUL *r*/m	Знаковое умножение (аналогично MUL). Все операнды считаются знаковыми. Команда IMUL имеет также двухоперандный и трехоперандный вид. Двухоперандный вид: `IMUL r,src` и `r<-r*src` Трехоперандный вид: `IMUL dst,src,imm` и `dst<-src*imm`
DIV *r/m* (*src*)	Беззнаковое деление. Аналогично беззнаковому умножению. Осуществляет деление аккумулятора и его расширения (AH:AL, DX:AX, EDX:EAX) на делитель *src*. Частное помещается в аккумуляторе, а остаток — в расширении аккумулятора
IDIV *r/m*	Знаковое деление. Аналогично беззнаковому
CBW	Расширение байта (AL) в слово с копированием знакового бита
CWD	Расширение слова (AX) в двойное слово (DX:AX) с копированием знакового бита

[5] Напоминаю, что ASCII-число предполагает одну цифру на один байт, BCD-число — одну цифру на половину байта. Таким образом, в регистре AX может находиться двухразрядное ASCII-число и четырехразрядное BCD-число.

Таблица 1.5 (окончание)

Команда	Описание
CWDE	Расширение слова (AX) в двойное слово (EAX) с копированием знакового бита
CDQ	Преобразование двойного слова (EAX) в учетверенное слово (EDX:EAX)

Таблица 1.6. *Логические операции*

Команда	Описание
AND *dest,src*	Логическая операция AND (И). Обнуление битов *dest*, которые равны нулю у *src*
TEST *dest,src*	Аналогична AND, но не меняет *dest*. Используется для проверки ненулевых битов
OR *dest,src*	Логическая операция OR (ИЛИ). В *dest* устанавливаются биты, отличные от нуля в *src*
XOR *dest,src*	Исключающее ИЛИ
NOT *dest*	Переключение всех битов (инверсия)

Таблица 1.7. *Сдвиговые операции*

Команда	Описание
RCL/RCR *dest,src*	Циклический сдвиг влево/вправо через бит переноса CF. *src* может быть либо CL, либо непосредственным операндом
ROL/ROR *dest,src*	Данная команда аналогична командам RCL/RCR, но иначе работает с флагом CF. Флаг CF не участвует в цикле сдвига, но в него попадает бит, перешедший с начала на конец или наоборот
SAL/SAR *dest,src*	Сдвиг влево/право. Называется еще *арифметическим сдвигом*. При сдвиге вправо дублируется старший бит. При сдвиге влево младший бит заполняется нулем. "Вытолкнутый" бит помещается в CF
SHL/SHR *dest,src*	Логический сдвиг влево/вправо. Сдвиг вправо отличается от SAR тем, что и старший бит заполняется нулем

Таблица 1.7 (окончание)

Команда	Описание
SHLD/SHRD *dest, src, count*	Трехоперандные команды сдвига влево/вправо. Первым операндом, как обычно, может быть либо регистр, либо ячейка памяти, вторым операндом должен быть регистр общего назначения, третьим — регистр CL или непосредственно операнд. Суть операции заключается в том, что *dest* и *src* вначале объединяются, а потом производится сдвиг на количество битов *count*. Результат снова помещается в *dest*

Таблица 1.8. *Строковые операции*

Команда	Описание
REP	Префикс, означающий повтор строковой операции до обнуления ECX. Префикс имеет также разновидности REPZ (REPE) — выполнять, пока не ноль (ZF=1), REPNZ (REPNE) — выполнять, пока ноль
MOVS *dest, src*	Команда передает байт, слово или двойное слово из цепочки, адресуемой DS:[ESI], в цепочку *dest*, адресуемую ES[EDI]. При этом EDI и ESI автоматически корректируются согласно значению флага DF. Допускается явная спецификация MOVSB (byte) — побайтовое копирование, MOVSW (word) — копирование словами, MOVSD (word) — четырехбайтовое копирование. *dest* и *src* можно явно не указывать
LODS *src*	Команда загрузки цепочки в аккумулятор. Имеет разновидности LODSB, LODSW, LODSD. При выполнении команды байт, слово, двойное слово загружается соответственно в AL, AX, EAX. При этом ESI автоматически изменяется на 1 в зависимости от значения флага DF. Префикс REP не используется
STOS *dest*	Команда, обратная LODS, т. е. передает байт, слово или двойное слово из аккумулятора в цепочку и автоматически корректирует EDI
SCAS *dest*	Команда сканирования цепочки. Команда вычитает элемент цепочки *dest* из содержимого аккумулятора (AL/AX/EAX) и модифицирует флаги. Префикс REPNE позволяет найти в цепочке нужный элемент

Таблица 1.8 *(окончание)*

Команда	Описание
CMPS *dest,src*	Команда сравнения цепочек. Данная команда производит вычитание байта, слова или двойного слова цепочки *dest* из соответствующего элемента цепочки *src*. В зависимости от результата вычитания модифицируются флаги. Регистры EDI и ESI автоматически продвигаются на следующий элемент. При использовании префикса REPE команда означает — сравнивать, пока не будет достигнут конец цепочки или пока элементы не будут равны. При использовании префикса REPNE команда означает — сравнивать, пока не достигнут конец цепочки или пока элементы будут равны

Таблица 1.9. *Команды управления флагами*

Команда	Описание
CLC	Сброс флага переноса
CMC	Инверсия флага переноса
STC	Установка флага переноса
CLD	Сброс флага направления
STD	Установка флага направления
CLI	Запрет маскируемых аппаратных прерываний
STI	Разрешение маскируемых аппаратных прерываний
CTS	Сброс флага переключения задач

Таблица 1.10. *Команды передачи управления*

Команда	Описание
JMP *target*	Имеет пять форм, различающихся расстоянием назначения от текущего адреса и способом задания целевого адреса. При работе в Windows используется в основном внутрисегментный переход (NEAR) в пределах 32-битного сегмента. Адрес перехода может задаваться непосредственно (в программе это метка) или косвенно, т. е. содержаться в ячейке памяти или регистре (JMP [EAX]). Другой тип перехода — короткий переход (SHORT), занимает всего 2 байта. Диапазон смещения, в пределах которого происходит переход: –128—127. Использование такого перехода весьма ограниченно.

Таблица 1.10 (продолжение)

Команда	Описание
	Межсегментный переход может иметь следующий вид: `JMP FWORD PTR L` где `L` — указатель на структуру, содержащую 48-битный адрес, в начале которого 32-й адрес смещения, затем 16-й селектор (сегмента, шлюза вызова, сегмента состояния задачи). Возможен также и такой вид межсегментного перехода: `JMP FWORD ES:[EDI]`
Условные переходы	• `JA/JNBE` — перейти, если выше; • `JAE/JNB` — перейти, если выше или равно; • `JB/JNAE` — перейти, если ниже; • `JBE/JNA` — перейти, если ниже; • `JC` — перейти, если перенос; • `JE/JZ` — перейти, если ноль; • `JG/JNLE` — перейти, если больше; • `JGE/JNL` — перейти, если больше или равно; • `JL/JNGE` — перейти, если меньше; • `JLE/JNG` — перейти, если меньше или равно; • `JNC` — перейти, если нет переноса; • `JNE/JNZ` — перейти, если меньше или равно; • `JNO` — перейти, если нет переполнения; • `JNP/JPO` — перейти, если нет паритета; • `JNS` — перейти, если нет знака; • `JO` — перейти, если есть переполнения; • `JP/JPE` — перейти, если есть паритет; • `JS` — перейти, если есть знак; • `JCXZ` — переход, если `CX=0`; • `JECXZ` — переход, если `ECX=0`. В плоской модели команды условного перехода осуществляют переход в пределах 32-битного регистра

Таблица 1.10 (окончание)

Команда	Описание
Команды управления циклом	Все команды этой группы уменьшают содержимое регистра ECX: • LOOP — переход, если содержимое ECX не равно нулю; • LOOPE (LOOPZ) — переход, если содержимое ECX не равно нулю и флаг ZF=1; • LOOPNE (LOOPNZ) — переход, если содержимое ECX не равно нулю и флаг ZF=0
CALL *target*	Передает управление процедуре (метке) с сохранением в стеке адреса, следующей за CALL-командой. В плоской модели адрес возврата представляет собой 32-битное смещение. Межсегментный вызов предполагает сохранение в стеке селектора и смещения, т. е. 48-битной величины (16 битов — селектор и 32 бита — смещение)
RET [*N*]	Возврат из процедуры. Необязательный параметр N предполагает, что команда также автоматически чистит стек (освобождает *N* байтов). Команда имеет разновидности, которые выбираются ассемблером автоматически, в зависимости от того, является процедура ближней или дальней. Можно, однако, и явно указать тип возврата (RETN или RETF). В случае плоской модели по умолчанию берется RETN с четырехбайтным адресом возврата

***Таблица 1.11.** Команды поддержки языков высокого уровня*

Команда	Описание
ENTER *par1,par2*	Подготовка стека при входе в процедуру. *par1* показывает количество байтов для локальных переменных в процедуре, *par2* — уровень вложенности в процедуру. При *par2*=0 вложенность процедур не допускается (ситуация в языке С)
LEAVE	Приведение стека в исходное состояние после выполнения команды ENTER
BOUND *r16,m16* или BOUND REG32,MEM32	Предполагается, что регистр содержит текущий индекс массива, а второй операнд определяет в памяти два слова или два двойных слова. Первое считается минимальным значением индекса, а второе — максимальным. Если текущий индекс оказывается вне границ, то генерируется команда INT 5. Используется для контроля нахождения индекса в заданных рамках, что является важным средством отладки

Таблица 1.12. *Команды прерываний*

Команда	**Описание**
INT *n*	Двухбайтная команда. Вначале в стек помещается содержимое регистра флагов, затем полный адрес возврата. Кроме того, сбрасывается флаг TF. После этого осуществляется косвенный переход через *n*-й элемент дескрипторной таблицы прерываний. Однобайтная команда INT 3 называется *прерыванием контрольного останова* и используется в программах-отладчиках
INTO	Равносильна команде INT 4, если флажок переполнения OF=1, если OF=0 — команда не производит никакого действия
IRET	Команда возврата из прерываний. Извлекает из стека сохраненные в нем адрес возврата и регистр флажков. Бит уровня привилегий будет модифицироваться только в том случае, если текущий уровень привилегий равен 0

Таблица 1.13. *Команды синхронизации процессора*

Команда	**Описание**
HLT	Останавливает процессор. Из такого останова процессор может быть выведен внешним прерыванием
LOCK	Представляет собой префикс блокировки шины. Он заставляет процессор сформировать сигнал LOCK# на время выполнения находящейся за префиксом команды. Этот сигнал блокирует запросы шины другими процессорами в мультипроцессорной системе
NOP	Холостая команда. Не производит никаких действий
WAIT (FWAIT)	Синхронизация с сопроцессором. Большинство команд сопроцессора автоматически вырабатывают эту команду

Таблица 1.14. *Команды обработки цепочки битов (появились в 386-м процессоре)*

Команда	**Описание**
BSF(BSR) *dest,src*	*dest* — 16-битный или 32-битный регистр. *src* — регистр или ячейка памяти. При выполнении команды BSF операнд *src* просматривается с младших, а в команде BSR — со старших битов. Номер первого встречного бита, находящегося в состоянии 1, помещается в регистр *dest*, флажок ZF сбрасывается в 0. Если *src* содержит 0, то ZF=1, а содержимое *dest* не определено

Таблица 1.14 *(окончание)*

Команда	Описание
BT *dest,src*	Тестирование бита с номером из *src* в *dest* и перенос его во флаг CF
BTC *dest,src*	Проверка и инвертирование бита из *src* в *dest*
BTR *dest,src*	Проверка и сброс бита из *src* в *dest*
BTS *dest,src*	Проверка и установка бита из *src* в *dest*

Таблица 1.15. *Команды управления защитой*

Команда	Описание
LGDT *src*	Загрузить GDTR из памяти. *src* указывает на 6-байтную величину
SGDT *dest*	Сохранить GDTR в памяти
LIDT *src*	Загрузить IDTR из памяти
SIDT *dest*	Сохранить IDTR в памяти
LLDT *src*	Загрузить LDTR из памяти (16 битов)
SLDT *dest*	Сохранить LDTR в регистре или памяти (16 битов)
LMSW *src*	Загрузка MSW
SMSW *dest*	Сохранить MSW в регистре или памяти (16 битов)
LTR *src*	Загрузка регистра задачи из регистра или памяти (16 битов)
STR *dest*	Сохранение регистра задачи в регистре или памяти (16 битов)
LAR *dest,src*	Загрузка старшего байта *dest* байтом прав доступа дескриптора *src*
LSL *dest,src*	Загрузка *dest* пределом сегмента, дескриптор которого задан *src*
ARPL *r/m,r*	Выравнивание RPL в селекторе до наибольшего числа из текущего уровня и заданного операндом
VERR *seg*	Верификация чтения: установка ZF=1, если задаче позволено чтение в сегменте *seg*
VERW *seg*	Верификация записи: установка ZF=1, если задаче позволена запись в сегменте *seg*

Таблица 1.16. *Команды обмена с управляющими регистрами*

Команда	Описание
MOV CR*n*,*src*	Загрузка управляющего регистра CR*n*
MOV *dest*,CR*n*	Чтение управляющего регистра CR*n*
MOV DR*n*,*src*	Загрузка регистра отладки DR*n*
MOV *dest*,DR*n*	Чтение регистра отладки DR*n*
MOV TR*n*,*src*	Загрузка регистра тестирования TR*n*
MOV *dest*,TR*n*	Чтение регистра тестирования TR*n*
RDTSC	Чтение счетчика тактов. Значение счетчика тактов помещается в пару регистров EDX:EAX

Таблица 1.17. *Команды идентификации и управления архитектурой*

Команда	Описание
CPUID	Получение информации о процессоре. Требует параметр в регистре EAX. Если EAX=0, процессор в регистрах EBX, EDX, ECX возвращает символьную строку, специфичную для производителя. Процессоры AMD возвращают строку "AuthenticAMD", процессоры Intel — "GenuineIntel". Если EAX=1, в младшем слове регистра EAX возвращает код идентификации. Если EAX=2, в регистрах EAX, EBX, ECX, EDX возвращаются параметры конфигурации процессора
RDMSR *r/m*	Чтение модельно-специфического регистра (MSR) в ECX
RDPMC	Помещает значение одного из двух программируемых счетчиков событий в пару регистров EDX:EAX. Выбор счетчика осуществляется по содержимому регистра ECX
WRMSR *r/m*	Запись ECX в модельно-специфический регистр
SYSENTER	Системный вызов
SYSEXIT	Возврат из системного вызова

Таблица 1.18. *Команды управления кэшированием*

Команда	Описание
INVD	Аннулирование данных в первичном кэше без обратной записи
WBINVD	Обратная запись модифицированных строк и аннулирование кэш-памяти

Таблица 1.18 (окончание)

Команда	Описание
`INVLPG r/m`	Аннулирование элемента таблицы трансляции TLB (TLB — буфер ассоциативной трансляции таблиц каталогов и страниц памяти)

1.2.3. Команды математического сопроцессора

А теперь мы коснемся основных положений работы арифметического сопроцессора.

До процессора Intel 80486 сопроцессор поставлялся отдельно, теперь он встраивается в процессор, являясь его неотъемлемой частью.

Функционирование и структура

Арифметический сопроцессор работает со своим набором команд и своим набором регистров. Однако выборку команд сопроцессора осуществляет процессор.

Арифметический сопроцессор выполняет операции со следующими типами данных: целое слово (16 битов), короткое целое (32 бита), длинное слово (64 бита), упакованное десятичное число (80 битов), короткое вещественное число (32 бита), длинное вещественное число (64 бита), расширенное вещественное число (80 битов). Форматы вещественных чисел были разобраны нами в *разд. 1.1.* Кроме обычных чисел в результате операций сопроцессора могут возникнуть также специальные случаи.

Специальные случаи

К ним относятся:

- положительный ноль (все биты нули);
- отрицательный ноль (знаковый бит равен 1);
- положительная бесконечность (знаковый бит 0, все биты мантиссы 0, все биты экспоненты 1);
- отрицательная бесконечность (знаковый бит 1, все биты мантиссы 0, все биты экспоненты 1);
- денормализованное число (все биты экспоненты 0);
- неопределенное число (знаковый бит 1, все биты экспоненты 1, первый бит мантиссы, а для 80-битного числа два бита 1, остальные 0);

- нечисловой экземпляр SNAN (все биты экспоненты 1, первый бит мантиссы 0, а для 80-битного числа первые два бита 10, а среди остальных битов есть 1);
- нечисловой экземпляр QNAN (все биты экспоненты 1, первый бит мантиссы, а для 80-битного числа два первых равны нулю, среди остальных битов мантиссы есть 1);
- неподдерживаемое число (ситуации, не соответствующие стандартным числам и не описанные в специальных случаях).

При выполнении операции сопроцессором процессор ждет завершения этой операции. Другими словами, перед каждой командой сопроцессора ассемблером автоматически генерируется команда, проверяющая, занят сопроцессор или нет. Если сопроцессор занят, процессор переводится в состояние ожидания. Иногда программисту требуется вручную ставить команду ожидания (WAIT) после команды сопроцессора.

Регистры данных

Сопроцессор имеет восемь 80-битных рабочих регистров, представляющих собой стековую кольцевую структуру. Эти регистры еще называют *стеком сопроцессора*. Регистры называются R0, R1—R7, но доступ к ним напрямую невозможен. Каждый регистр может занимать любое положение в стеке. Название стековых (относительных) регистров: ST(0), ST(1), ST(2), ST(3), ST(4), ST(5), ST(6), ST(7).

Кроме того, имеется еще регистр состояния (слово состояния) SW, по флагам которого можно, в частности, судить о результате выполненной операции. Регистр управления (слово управления) CW содержит в себе биты, влияющие на выполнение команд сопроцессора.

Регистр тегов TW содержит 16 битов, описывающих содержание регистров сопроцессора — по два бита на каждый рабочий регистр. Тег говорит о содержимом регистра данных. Вот значение тегов: 00 — действительное ненулевое число, 01 — истинный ноль, 10 — специальные числа, 11 — отсутствие данных.

Кроме указанных регистров сопроцессор имеет также регистры FIP и EIP. Регистр FIP содержит адрес последней выполняемой команды, кроме FINIT, FCLEX, FLDCW, FSTCW, FSTSW, FSTSWAX, FSTENV, FLDENV, FSAVE, FRSTOR, FWAIT. Регистр FDP содержит адрес операнда команды, кроме указанных выше.

При вычислении с помощью команд сопроцессора большую роль играют *исключения* или *особые ситуации*. Типичной особой ситуацией является деление на 0. Биты особых ситуаций хранятся в регистре состояний. Учет особых ситуаций необходим для получения правильных результатов.

Список особых ситуаций

К особым ситуациям относятся следующие:

- неточный результат (округление);
- недействительная операция;
- деление на ноль;
- антипереполнение (слишком маленький результат);
- переполнение (слишком большой результат);
- денормализованный операнд.

Слово состояния *SW* (Status Word)

Слово состояния арифметического сопроцессора отражает его общее состояние:

- 0-й бит, флаг недопустимой операции (`IE`);
- 1-й бит, флаг денормализованной операции (`DE`);
- 2-й бит, флаг деления на ноль (`ZE`);
- 3-й бит, флаг переполнения (`OE`);
- 4-й бит, флаг антипереполнения (`UE`);
- 5-й бит, флаг неточного результата (`PE`);
- 6-й бит, ошибка стека (`SF`);
- 7-й бит, общий флаг ошибки (`ES`);
- 8—10, 14-й, флаги условий (`C0`, `C1`, `C2`, `C3`);
- 11—13-й, число (0—7), показывающее, какой регистр является вершиной;
- 15-й бит, флаг занятости, совпадает с `ES`.

Слово управления *CW* (Control Word)

Слово управления арифметического сопроцессора определяет один из нескольких вариантов обработки численных данных:

- 0-й бит, маска недействительной операции;
- 1-й бит, маска денормализованного операнда;
- 2-й бит, маска деления на ноль;
- 3-й бит, маска переполнения;
- 4-й бит, маска антипереполнения;
- 5-й бит, маска неточного результата;

- 6-й, 7-й биты, резерв;
- 8-й, 9-й биты, управление точностью;
- 10-й, 11-й биты, управление округлением;
- 12-й, управление бесконечностью;
- 13—15-й, резерв.

Возможные причины исключений:

- ошибка стека; результат — неопределенное число;
- операция с неподдерживаемым числом; результат — неопределенное число;
- операция с SNAN; результат — QNAN;
- сравнение числа с QNAN или SNAN; результат — C0 = C2 = C3 = 1;
- Сложение бесконечностей с одним или вычитание с разными знаками; результат — неопределенное число;
- умножение нуля на бесконечность; результат — неопределенное число;
- деление бесконечности на бесконечность или 0 на 0; результат — неопределенное число;
- команды FPREM и FPREM1, если делитель равен 0 или делимое равно бесконечности; результат — неопределенное число и C2 = 0;
- тригонометрические операции над бесконечностью; результат — неопределенное число и C2 = 0;
- корень или логарифм, если аргумент меньше нуля; результат — неопределенное число;
- FBSTP, если регистр-источник пуст, содержит QNAN или SNAN, бесконечность или превышает 18 десятичных знаков; результат — неопределенное число;
- FXCH, если один из операндов пуст; результат — неопределенное число.

Команды сопроцессора

В табл. 1.19—1.23 дан полный перечень команд арифметического сопроцессора с пояснением операций, которые они выполняют.

***Таблица 1.19.** Команды передачи данных*

Команда	Описание
FLD *src*	Загрузить вещественное число в ST(0) (вершину стека) из области памяти. При этом ST(0)->ST(1). Область памяти может быть 32-, 64-, 80-битной. Команда FLD ST(0) дублирует вершину стека

Таблица 1.19 (окончание)

Команда	Описание
FILD *src*	Загрузить целое число в ST(0) из памяти. При этом ST(0)->ST(1). Область памяти может быть 16-, 32-, 64-битной
FBLD *src*	Загрузить BCD-число в ST(0) из 80-битной области памяти
FLDZ	Загрузить 0 в ST(0)
FLD1	Загрузить 1 в ST(0)
FLDPI	Загрузить PI в ST(0)
FLDL2T	Загрузить LOG2(10) в ST(0)
FLDTL2E	Загрузить LOG2(e) в ST(0)
FLDLG2	Загрузить LG(2) в ST(0)
FLDLN2	Загрузить LN(2) в ST(0)
FST *dest*	Записать вещественное число из ST(0) в память. Область памяти может быть 32-, 64- или 80-битной
FSTP *dest*	Записать вещественное число из ST(0) в память. Область памяти может быть 32-, 64- или 80-битной. При этом происходит выталкивание вершины из стека
FBST *dest*	Записать BCD-число в память. Область памяти — 80-битная
FBSTP *dest*	Записать BCD-число в память. Область памяти — 80-битная. При этом происходит выталкивание вершины из стека
FXCH ST(*i*)	Выполнить обмен значениями вершины стека и регистра *i*. Если операнд не указан, то обмениваются ST(0) и ST(1)
FCMOVc *dest*,*src*	Команда условной пересылки данных. Копирование ST(*i*) (*src*) в ST(0) (*dest*). Команда может иметь следующий вид: • FCMOVE — копировать, если равно (ZF=1); • FCMOVNE — копировать, если не равно (ZF=0); • FCMOVB — копировать, если меньше (CF=1); • FCMOVBE — копировать, если меньше или равно (CF=1 и ZF=1); • FCMOVNB — копировать, если меньше (CF=0); • FCMOVNBE — копировать, если меньше или равно (CF=0 и ZF=1); • FCMOVU — копировать, если не сравнимы (PF=1); • FCMOVNU — копировать, если сравнимы (PF=0)

Таблица 1.20. Команды сравнения данных

Команда	Описание
FCOM	Сравнение вещественных чисел ST(0) и ST(1). Флаги устанавливаются, как при операции вычитания ST(0)-ST(1). В этой команде и далее (до команды FCOMI) команды следующим образом влияют на флаги C0, C2, C3: • ST(0) > src C0=0, C2=0, C3=0; • ST(0) < src C0=1, C2=0, C3=0; • ST(0) = src C0=0, C2=0, C3=1. Если операнды несравнимы, то C0=C2=C3=1
FCOM *src*	Сравнение ST(0) с операндом в памяти. Операнд может быть 32- или 64-битным
FCOMP *src*	Сравнение вещественного числа в ST(0) с операндом с выталкиванием ST(0) из стека. Операнд может быть регистром и областью памяти
FCOMPP	Сравнение ST(0) и ST(1) с двойным выталкиванием из стека
FICOM *src*	Сравнение целых чисел в ST(0) с операндом. Операнд может быть 16- или 32-битным
FICOMP *src*	Сравнение целых чисел в ST(0) с операндом. Операнд может быть 16- или 32-битной областью памяти или регистром. При выполнении операции происходит выталкивание ST(0) из стека
FTST	Проверка ST(0) на ноль
FUCOM ST(*i*)	Сравнение ST(0) с ST(*i*) без учета порядков
FUCOMP ST(*i*)	Сравнение ST(0) с ST(*i*) без учета порядков. При выполнении операции происходит выталкивание из стека
FUCOMPP ST(*i*)	Сравнение ST(0) с ST(*i*) без учета порядков. При выполнении операции происходит двойное выталкивание из стека
FCOMI *src*	Сравнить и установить флаги. Команда FCOMI и следующие за ней команды FCOMIP, FUCOMI, FUCOMIP воздействуют на биты регистра флагов следующим образом: • ST(0) > src ZF=0, PF=0, CF=0; • ST(0) < src ZF=0, PF=0, CF=1; • ST(0) = src ZF=1, PF=0, CF=0. Если операнды несравнимы, то все три флага равны 1

Таблица 1.20 (окончание)

Команда	Описание
FCOMIP *src*	Сравнить, установить биты и вытолкнуть
FUCOMI *src*	Сравнить без учета порядков и установить флаги
FUCOMIP *src*	Сравнить без учета порядков, установить флаги и вытолкнуть
FXAM	Анализ содержимого вершины стека. Результат помещается в биты C3, C2, C0 следующим образом: • 000 — неподдерживаемый формат; • 001 — не число; • 010 — нормализованное число; • 011 — бесконечность; • 100 — ноль; • 101 — пустой операнд; • 110 — денормализованное число

Таблица 1.21. Арифметические команды

Команда	Описание
FADD *src* FADD ST(*i*),ST	Сложение вещественных чисел: ST(0)<-ST(0)+*src* где *src* — 32- или 64-битное число ST(i)<-ST(i)+ST(0)
FADDP ST(*i*),ST	Сложение вещественных чисел: ST(*i*)<-ST(*i*)+ST(0) При выполнении операции происходит выталкивание из стека
FIADD *src*	Сложение целых чисел: ST(0)<-ST(0)+*src* где src — 16- или 32-битное число
FSUB *src* FSUB ST(*i*),ST	Вычитание вещественных чисел: ST(0)<-ST(0)-*src* где *src* — 32- или 64-битное число ST(i)<-ST(i)-ST(0)

Таблица 1.21 (продолжение)

Команда	Описание
`FSUBP ST(i),ST`	Вычитание вещественных чисел: `ST(i)<-ST(i)-ST(0)` При выполнении операции происходит выталкивание стека
`FSUBR ST(i),ST`	Обратное вычитание вещественных чисел: `ST(0)<-ST(i)-ST(0)`
`FSUBRP ST(i),ST`	Обратное вычитание вещественных чисел: `ST(0)<-ST(i)-ST(0)` При выполнении операции происходит выталкивание из стека
`FISUB src`	Вычитание целых чисел: `ST(0)<-ST(0)-src` где *src* — 16- или 32-битное число
`FISUBR src`	Вычитание целых чисел: `ST(0)<-ST(0)-src` где *src* — 16- или 32-битное число. При выполнении операции происходит выталкивание из стека
`FMUL` `FMUL ST(i)` `FMUL ST(i),ST`	Умножение двух операндов. В первом случае `ST(0)<-ST(0)*ST(1)`. Во втором случае `ST(0)<-ST(i)*ST(0)`. В третьем случае `ST(i)<-ST(i)*ST(0)`
`FMULP ST(i),ST(0)`	Умножение и выталкивание из стека: `ST(i)<-ST(i)*ST(0)`
`FIMUL src`	Умножение `ST(0)` на целое число: `ST(0)<-ST(0)*src` Операнд может быть 16- и 32-битным числом
`FDIV` `FDIV ST(i)` `FDIV ST(i),ST`	Деление `ST(0)`: `ST(0)<-ST(0)/ST(1)` `ST(0)<-ST(0)/ST(i)` `ST(i)<-ST(0)/ST(i)`
`FDIVP ST(i),ST`	Деление с выталкиванием из стека: `ST(i)<-ST(0)/ST(i)`

Таблица 1.21 (окончание)

Команда	Описание
FIDIV *src*	Деление целых чисел: ST(0)<-ST(*i*)/*src* Делитель может быть 16- и 32-битным числом
FDIVR ST(*i*),ST	Обратное деление вещественных чисел: ST(0)<-ST(*i*)/ST(0)
FDIVRP ST(*i*),ST	Обратное деление вещественных чисел и выталкивание из стека: ST(0)<-ST(*i*)/ST(0)
FIDIVR *src*	Обратное деление целых чисел: ST(0)<-*src*/ST(0)
FSQRT	Извлечь корень из ST(0) и поместить обратно
FSCALE	Масштабирование: ST(0)<-ST(0)*2^ST(1)
FXTRACT	Выделение мантиссы и порядка из числа ST(0). В ST(0) помещается порядок, в ST(1) — мантисса
FPREM	Нахождение остатка от деления: ST(0)<-ST(0)MODST(1)
FPREM1	Нахождение остатка от деления в стандарте IEEE[6]
FRNDINT	Округление до ближайшего целого числа, находящегося в ST(0): ST(0)<-int(ST(0))
FABS	Нахождение абсолютного значения: ST(0)<-ABS(ST(0))
FCSH	Изменение знака: ST(0)<- -ST(0)

***Таблица 1.22.** Трансцендентные функции*

Команда	Описание
FCOS	Вычисление косинуса: ST(0)<-COS(ST(0)) Содержимое в ST(0) интерпретируется как угол в радианах

[6] Institute of Electrical and Electronics Engineers, Институт инженеров по электротехнике и электронике.

***Таблица 1.22** (окончание)*

Команда	Описание
FPTAN	Частичный тангенс. Содержимое в ST(0) интерпретируется как угол в радианах. Значение тангенса возвращается на место аргумента, а затем в стек включается 1
FPATAN	Вычисление арктангенса. Вычисляется функция Arctg(ST(1)/ST(0)) После вычисления происходит выталкивание из стека, в итоге результат оказывается в вершине стека
FSIN	Вычисление синуса: ST(0)<-SIN(ST(0)) Содержимое в ST(0) интерпретируется как угол в радианах
FSINCOS	Вычисление синуса и косинуса: ST(0)<-SIN(ST(0)) ST(1)<-COS(ST(0))
F2XM1	Вычисление $2^x - 1$: ST(0)<-2^ST(0)-1
FYL2X	Вычисление $y \times \log_2 x$: ST(0)=Y ST(1)=X Происходит выталкивание из стека, и только потом в вершину стека помещается результат вычисления
FYL2XP1	Вычисление $y \times \log_2(x + 1)$: ST(0)=Y ST(1)=X Происходит выталкивание из стека, и только потом в вершину стека помещается результат вычисления

***Таблица 1.23.** Команды управления сопроцессором*

Команда	Описание
FINIT	Инициализация сопроцессора
FNINIT	Инициализация сопроцессора без ожидания
FSTSW AX	Запись слова состояния в AX (SW->AX)
FSTSW *dest*	Запись слова состояния в *dest* (16 битов)

Таблица 1.23 (окончание)

Команда	Описание
FNSTSW *dest*	Сохранить слово состояния без ожидания в *dest* (16 бит)
FLDCW *src*	Загрузка управляющего слова (16 битов) из *dest*
FSTCW *dest*	Сохранение управляющего слова в *dest*
FCLEX	Сброс исключений
FNCLEX	Сброс исключений без ожидания
FSTENV *dest*	Сохранение состояния сопроцессора (SW, CW, TAGW, FIP, FDP) в памяти
FNSTENV *dest*	Сохранение состояния сопроцессора (SW, CW, TAGW, FIP, FDP) в памяти без ожидания
FLDENV *src*	Загрузка состояния сопроцессора из памяти
FSAVE *dest*	Сохранение состояния сопроцессора и файла регистров в памяти
FNSAVE *dest*	Сохранение состояния сопроцессора и файла регистров в памяти без ожидания
FRSTOR *src*	Загрузка состояния сопроцессора и файла регистров в память
FINCSTP	Инкремент указателя стека
FDECSTP	Декремент указателя стека
FFREE ST(*i*)	Освобождение регистра — пометка ST(*i*) как свободного
FNOP	Холостая операция сопроцессора
WAIT (FWAIT)	Ожидание процессором завершения текущей операции сопроцессора

1.2.4. Расширение MMX

Архитектура MMX

Расширение MMX в основном ориентировано на использование в мультимедийных приложениях. Главная идея MMX заключается в одновременной обработке нескольких элементов данных за одну инструкцию. Расширение MMX появилось в процессорах модификации Pentium P54C и присутствует во всех последних модификациях этого процессора.

Расширение MMX использует новые типы упакованных данных: упакованные байты (восемь байтов), упакованные слова (четыре слова), упакованные двойные слова (два двойных слова), учетверенное слово. Как видим, все это

64-битные числа. Расширение MMX включает восемь регистров общего пользования (обозначения `MM0`—`MM7`). Размер регистров составляет 64 бита. Физически эти регистры пользуются младшими битами рабочих регистров сопроцессора (`R0`—`R7`). Команды MMX "портят" регистр состояния и регистр тегов. По этой причине совместное использование команд MMX и команд сопроцессора может вызвать определенные трудности. Другими словами, перед каждым применением команд MMX вам придется сохранять контекст сопроцессора, а это может весьма замедлить работу программы. Важно отметить также, что команды MMX работают непосредственно с регистрами сопроцессора, а не с указателями на элементы стека.

Инструкции MMX

В табл. 1.24 представлены команды расширения MMX.

Таблица 1.24. *Команды расширения MMX*

Команда	Описание
EMMS	Очистка стека регистров. Установка всех единиц в слове тегов
MOVD *mm,m32/ir32*	Пересылка данных в младшие 32 бита регистра MMX с заполнением старших битов нулями
MOVD *m32/ir32,mm*	Пересылка данных из младших 32 битов регистра MMX
MOVQ *mm,mm/m64*	Пересылка данных в регистр MMX
MOVQ *mm/m64,mm*	Пересылка данных из регистра MMX
PACKSSDW *mm,mm/m64*	Упаковка со знаковым насыщением двух двойных слов, расположенных в *mm*, и двух двойных слов *mm/m64* в четыре слова, расположенных в *mm*. Другими словами, команда копирует два двойных слова из *mm* в два младших слова *mm* и два двойных слова из *mm/m64* в два старших слова. При этом если значение какого-либо двойного слова окажется больше 32 767 или меньше –32 768, то в слова запишутся 32 767 и –32 768 соответственно
PACKSSWB *mm,mm/m64*	Упаковка со знаковым насыщением четырех слов, расположенных в *mm*, и четырех слов *mm/m64* в восемь байтов, расположенных в mm. Другими словами, четыре слова из *mm* преобразуются в четыре младших байта *mm*, а четыре слова из *mm/m64* — в четыре старших байта. При этом если значение какого-либо слова окажется больше 127 или меньше –128, то в байты помещаются соответственно числа 127 и –128

Таблица 1.24 (продолжение)

Команда	Описание
PACKUSWB *mm*, *mm/m64*	Упаковка с насыщением четырех знаковых слов, расположенных в *mm*, и четырех слов *mm/m64* в восемь беззнаковых байтов, расположенных в *mm*. Другими словами, четыре слова из *mm* преобразуются в четыре младших байта *mm*, а четыре слова из *mm/m64* — в четыре старших байта. При этом если значение какого-либо слова окажется больше 255 или меньше 0, то в байты помещаются соответственно числа 255 и 0
PADDB *mm*, *mm/m64* PADDW *mm*, *mm/m64* PADDD *mm*, *mm/m64*	Сложение упакованных байтов (слов или двойных слов) без насыщения (с циклическим переполнением). Если при сложении возникает перенос, то он не влияет ни на следующие элементы, ни на флаг переноса, т. е. просто игнорируется
PADDSB *mm*, *mm/m64* PADDSW *mm*, *mm/m64*	Сложение упакованных байтов (слов) со знаковым насыщением
PADDUSB *mm*, *mm/m64* PADDUSW *mm*, *mm/m64*	Сложение упакованных байтов (слов) с беззнаковым насыщением
PAND *mm*, *mm/m64*	Логическое "И"
PANDN *mm*, *mm/m64*	Логическое "И-НЕ"
PCMPEQB *mm*, *mm/m64* PCMPEQD *mm*, *mm/m64* PCMPEQW *mm*, *mm/m64*	Сравнение на равенство упакованных байтов (слов, двойных слов). Все биты элемента результата будут единичными (`true`) при совпадении соответствующих элементов операндов и нулевыми (`false`) — при несовпадении
PCMPGTB *mm*, *mm/m64* PCMPGTD *mm*, *mm/m64* PCMPGTW *mm*, *mm/m64*	Сравнение по величине упакованных знаковых байтов (слов, двойных слов). Все биты элемента результата будут единичными (`true`), если соответствующий элемент операнда назначения больше элемента операнда источника, и нулевыми (`false`) в противном случае
PMADDWD mm, mm/m64	Умножение четырех знаковых слов операнда источника на четыре знаковых слова операнда назначения. Два двойных слова результатов умножения младших слов суммируются и записываются в младшее двойное слово операнда назначения. Два двойных слова результатов умножения старших слов суммируются и записываются в старшее двойное слово операнда назначения
PMULHW *mm*, *mm/m64*	Умножение упакованных знаковых слов с сохранением только старших 16 битов элементов результата

Таблица 1.24 (продолжение)

Команда	Описание
PMULLW *mm, mm/m64*	Умножение упакованных знаковых или беззнаковых слов с сохранением только младших 16 битов результата
POR *mm, mm/m64*	Логическое "ИЛИ"
PSHIMD *mm, imm* PSHIMQ *mm, imm* PSHIMW *mm, imm*	PSHIMD представляет инструкции PSLLD, PSRAD и PSRLD с непосредственным операндом-счетчиком PSHIMW представляет инструкции PSLLW, PSRAW, PSRLW. PSHIMQ представляет инструкции PSLLQ и PSRLQ с непосредственным операндом-счетчиком
PSLLD *mm, mm/m64* PSLLQ *mm, mm/m64* PSLLW *mm, mm/m64*	Логический сдвиг влево упакованных слов (двойных, учетверенных) операнда назначения на количество битов, указанных в операнде-источнике, с заполнением младших битов нулями
PSRAD *mm, mm/m64* PSRAW *mm, mm/m64*	Арифметический сдвиг вправо упакованных двойных (учетверенных) знаковых слов операнда назначения на количество битов, указанных в операнде-источнике, с заполнением младших битов битами знаковых разрядов
PSRLD *mm, mm/m64* PSRLQ *mm, mm/m64* PSRLW *mm, mm/m64*	Логический сдвиг вправо упакованных слов (двойных, учетверенных) операнда назначения на количество битов, указанных в операнде-источнике, с заполнением старших битов нулями
PSUBB *mm, mm/m64* PSUBW *mm, mm/m64* PSUBD *mm, mm/m64*	Вычитание упакованных байтов (слов или двойных слов) без насыщения (с циклическим антипереполнением)
PSUBSB *mm, mm/m64* PSUBSW *mm, mm/m64*	Вычитание упакованных знаковых байтов (слов) с насыщением. Если при вычитании разность выходит за пределы байта (слова), то в качестве результата используется минимальное число
PSUBUSB *mm, mm/m64* PSUBUSW *mm, mm/m64*	Вычитание упакованных беззнаковых байтов (слов) с насыщением. Если при вычитании разность выходит за пределы результата, то в качестве результата используется минимальное число (0)
PUNPCKHBW *mm, mm/m64*	Чередование в регистре назначения байтов старшей половины операнда-источника с байтами старшей половины операнда назначения

Таблица 1.24 (окончание)

Команда	Описание
PUNPCKHWD *mm,mm/m64*	Чередование в регистре назначения слов старшей половины операнда-источника со словами старшей половины операнда назначения
PUNPCKHDQ *mm,mm/m64*	Чередование в регистре назначения двойного слова старшей половины операнда-источника с двойным словом старшей половины операнда назначения
PUNPCKLBW *mm,mm/m64*	Чередование в регистре назначения байтов младшей половины операнда-источника с байтами младшей половины операнда назначения
PUNPCKLWD *mm,mm/m64*	Чередование в регистре назначения слов младшей половины операнда-источника со словами младшей половины операнда назначения
PUNPCKLDQ *mm,mm/m64*	Чередование в регистре назначения двойного слова младшей половины операнда-источника с двойным словом младшей половины операнда назначения
PXOR *mm,mm/m64*	Исключающее "ИЛИ"

Новые инструкции MMX

Перечисленные инструкции группы MMX с появлением Pentium 4 получили доступ к 128-битным регистрам (*xmm*). В табл. 1.25 перечислены новые MMX-инструкции. В командах табл. 1.25 первым идет операнд-получатель (dest), вторым — источник (src).

Таблица 1.25. Новые команды MMX

Команда	Описание
PADDQ *xmm,xmm/m128*	Сложение двух 128-битных операндов
PSUBQ *xmm,xmm/m128*	Вычитание 128-битных операндов
PMULUDQ *xmm,xmm/m128*	Умножение 64-битных операндов, результат не должен превышать 128-битный размер
PSLLDQ *xmm,imm*	Логический сдвиг содержимого влево на imm*8 битов
PSRLDQ *xmm,imm*	Логический сдвиг содержимого вправо на imm*8 битов
PSHUFHW *xmm,xmm/m128,imm*	Пересылка (из *xmm/m128* — *src* в *xmm* — *dest*) с перегруппировкой четырех 16-битных слов из младшей половины *dest* в младшую половину *src*. Перегруппировка задается содержимым константы *imm*

Таблица 1.25 (окончание)

Команда	Описание
PSHUFLW *xmm*, *xmm/m128*, *imm*	Пересылка (из *xmm/m128* — *src* в *xmm* — *dest*) с перегруппировкой четырех 16-битных слов из старшей половины *dest* в старшую половину *src*. Перегруппировка задается содержимым константы *imm*
PSHUFD *xmm*, *xmm/m128*, *imm*	Пересылка (из *xmm/m128* — *src* в *xmm* — *dest*) с перегруппировкой четырех 32-битных слов из *dest* в *src*. Перегруппировка задается содержимым константы *imm*
PUNPCKHQDQ *xmm*, *xmm/m128*	В *dest* (*xmm*) записывается содержимое старших половин *src* (*xmm/m128*) и *dest*
PUNPCKLQDQ *xmm*, *xmm/m128*	В *dest* (*xmm*) записывается содержимое младших половин *src* (*xmm/m128*) и *dest*
MOVDQ2Q *mm*, *xmm*	Младшая половина *xmm* копируется в *mm*
MOV2QDQ *xmm*, *mm*	Содержимое регистра *mm* копируется в младшую половину *xmm*
MOVNTDQ *m128*, *xmm*	Пересылка (из xmm - src в m128 - dest) содержимого 128-битного регистра в память без кэширования. Адрес должен быть кратен 16
MOVDQA *xmm*, *xmm/m128* MOVDQA *xmm/m128*, *xmm*	Команды пересылки 128-битного кода. Данные в памяти должны иметь адрес, кратный 16
MOVDQU *xmm*, *xmm/m128* MOVDQU *xmm/m128*, *xmm*	Команды пересылки 128-битного кода. Данные в памяти могут не иметь 16-битного выравнивания
MOVMSKPD *r32*, *xmm*	Копирует содержимое знаковых разрядов (63 и 127) в биты 0 и 1 регистра r32. Остальные биты регистра очищаются
MASKMOVDQU *xmm*, *xmm*	Пересылка по маске. Первый операнд *xmm* (*src*) содержит пересылаемый код, второй операнд *xmm* (*dest*) — маску пересылки. Адрес третьего (куда будет производиться пересылка) операнда должен находиться в DS:DI или в DS:EDI. Для каждого из 16-ти байтов выполняется следующее: если знаковый разряд i-го байта маски установлен, то *dest*[*i*]=*src*[*i*]; если знаковый разряд байта маски очищен, то содержимое *dest* не изменяется

1.3. Особенности программирования в операционной системе Windows

Сейчас мы намерены дать краткое введение в программирование в среде Windows. Это нельзя назвать курсом, поскольку для этого потребовалась бы целая книга. Я просто хочу напомнить читателям основные принципы программирования под Windows, которые, я надеюсь, пригодятся в дальнейшем при анализе исполняемых модулей.

1.3.1. Общие положения

Программирование в Windows основывается на использовании функций *API*. API (Application Program Interface) — это программный интерфейс приложения. С помощью API-функций приложение может взаимодействовать непосредственно с операционной системой Windows. Приложение, построенное на таком взаимодействии, более тесно интегрировано в операционную систему, а значит, обладает большими, по сравнению с другими программами, возможностями. Иногда API-функции называют *системными вызовами*. Но это не совсем точно. Дело в том, что системный вызов, как, например, в UNIX, — это вызов системной процедуры, хранящейся в ядре. Операционная система предоставляет ряд таких процедур для того, чтобы облегчить прикладной программе управление ресурсами. Функции API — это интерфейс между системными процедурами и прикладной программой, дополнительный слой. При вызове функции API вы не знаете, будет ли она выполняться полностью за счет кода загруженной в ваше адресное пространство динамической библиотеки или все-таки использует какие-то процедуры, хранимые в ядре. Операционная система Windows меняется, появляются новые версии, а интерфейс API остается тем же, в него могут добавляться лишь новые функции. Этим достигается полная совместимость программ, написанных с использованием базового набора функций API.

Подключение API-функций осуществляется посредством динамических библиотек, хранящихся в системном каталоге (Windows\system32). Подключение этих библиотек обеспечивает компилятор (позднее неявное связывание). Полное количество всевозможных API-функций огромно и превосходит 3000. Чаще всего употребляются API-функции, расположенные в следующих четырех динамических библиотеках:

- kernel32.dll — здесь содержатся функции управления (памятью, приложениями, ресурсами, файлами и т. д.);
- user32.dll — содержит всевозможные функции программного интерфейса пользователя (обработка оконных сообщений, таймеры, меню и т. д.);

- gdi32.dll — библиотека интерфейса графического устройства (Graphic Device Library), содержит большое количество функций оконной графики;
- comctl32.dll — данная библиотека содержит функции обслуживания различных управляющих элементов. В частности, именно эта библиотека отвечает за новый стиль управляющих элементов (стиль Windows XP).

Если у функции API один из входных параметров — указатель на строку, то такая функция имеет две версии: с префиксом A для строк в кодировке ANSI и с префиксом W для строк в кодировке Unicode. Например, имеются две версии для функции API `MessageBox`: `MessageBoxA` и `MessageBoxW`. В случае языков высокого уровня, например, языка C++, мы изначально определяем, с какой строкой или строками мы работаем, поэтому компилятор автоматически выбирает соответствующую версию функции. При написании программы на языке ассемблера мы явно указываем, какую версию функции используем.

Рассмотрим два типа приложений, используемых для прикладного программирования, — *консольные приложения* и *приложения GUI*. Особенностью консольного приложения является то, что при его выполнении система создает для него (или приложение наследует от порождающего его процесса) текстовое (консольное) окно или просто *консоль*. GUI (Graphic User Interface) — это графический интерфейс пользователя. Приложения, построенные по данной схеме, работают с графическими окнами, которые могут содержать графические изображения, а также различные элементы управления (кнопки, поля ввода, списки и т. д.). Приложения GUI мы будем называть также *графическими* или *оконными приложениями*.

В Windows могут выполняться и другие типы приложений — это службы (services) и драйверы. Их правильнее было бы называть системными типами приложений. Кроме этого, в Windows можно исполнять приложения в подсистемах POSIX[7] и OS/2, но с весьма ограниченными возможностями. Эти типы приложений не будут являться предметом изучения данной книги.

Довольно часто для написания используют библиотечные функции (C, C++) или библиотечные классы (в Delphi их называют компонентами). Тогда взаимодействие с операционной системой оказывается скрытым слоем библиотек. В результате анализ исполняемого кода усложняется, т. к. надо либо распознать, какая известная библиотечная функция или класс используется, либо путем анализа библиотечного кода выяснить, к каким функциям API происходит обращение, и попытаться понять цель этих обращений. Обе задачи не просты. Цель данного раздела — объяснить общую структуру программы для Windows, чтобы вы могли понять подходы в решении второй задачи.

---

[7] Portable Operating System Interface — переносимый интерфейс операционной системы.

По сути, все различие консольного и графического приложений для операционной системы заключается во флаге `Subsystem`, который хранится в заголовке `PE` *(см. разд. 1.6)*. Флаг же устанавливается при компоновке приложения. Для редактора связей LINK.EXE указывается ключ /SUBSYSTEM:WINDOWS — для графического приложения и ключ /SUBSYSTEM:CONSOLE — для консольного приложения. Соответственно в случае языка высокого уровня у компилятора должны быть опции, позволяющие создавать консольные или графические приложения. При этом и консольные, и графические приложения абсолютно равны в праве доступа к ресурсам операционной системы. Любое консольное приложение может создавать графические окна и работать с ними, в свою очередь графическое приложение способно работать и с текстовым консольным окном.

1.3.2. Консольные приложения

Консольные приложения очень компактны не только в откомпилированном виде, но и в текстовом варианте. Несколько слов следует сказать и о самой консоли. Как вы, наверное, уже знаете, консоль — это текстовое окно. Взаимоотношение консольной программы с таким окном сводится к следующему:

- если консольная программа запущена другой консольной программой, то дочерняя программа по умолчанию будет наследовать консоль родительской;
- если родительская программа не имеет консоли, то система создает для вновь запускаемой программы собственную консоль;
- консольная программа может создать и свою консоль с помощью API-функции `AllocConsole`, если предварительно освободится от уже имеющейся у нее консоли;
- консольная программа может иметь только одну консоль (см. предыдущий пункт).

Одной из причин, по которой консоль появилась в изначально графически ориентированной операционной системе Windows, являлась необходимость выполнения в ней программ, написанных для операционной системы MS-DOS, которая, как известно, была ориентирована на работу с текстовым экраном. При запуске такой программы операционная система Windows автоматически выделяет для нее консоль и автоматически же перенаправляет весь ввод/вывод туда.

Классическую структуру консольного приложения можно назвать пакетной. Программа состоит из последовательности выполняемых действий. Например, программа открывает некоторый файл, делает что-то, закрывает его и заканчивает свою работу.

В листинге 1.2 представлена типичная консольная программа[8], которая выводит на текстовый экран строку. Особенностью данной программы является то, что она создает свою консоль в независимости от того, запущена она сама из консоли или другим способом. Последовательность вызовов функций FreeConsole()/AllocConsole() освобождает имеющуюся у программы консоль и создает новую консоль. С унаследованной же консолью ничего не происходит, программа просто получает возможность создавать свою консоль. Если вы уберете функцию FreeConsole() в начале программы и запустите ее из консольного приложения, то новая консоль создаваться не будет, а весь вывод произойдет в существующую консоль, несмотря на наличие функции AllocConsole().

Листинг 1.2

```
#include <windows.h>
char * s="Example of console program.\n\0";
char buf[100];
DWORD d;
void main()
{
//освободить консоль, если она унаследована
  FreeConsole();
//создать консоль
  AllocConsole();
//получить дескриптор вывода на консоль
  HANDLE ho=GetStdHandle(STD_OUTPUT_HANDLE);
//получить дескриптор ввода с консоли
  HANDLE hi=GetStdHandle(STD_INPUT_HANDLE);
//вывести строку на консоль
  WriteConsole(ho,s,lstrlen(s),&d,NULL);
//использовать ReadConsole для просмотра экрана консоли
  ReadConsole(hi,(void*)buf,100,&d,NULL);
//закрыть дескрипторы
  CloseHandle(ho);
  CloseHandle(hi);
//освободить консоль
  FreeConsole();
}
```

[8] Здесь и далее программы на C++, если это не будет оговорено особо, были созданы в среде Visual Studio .NET.

3 Зак. 119

Программа из листинга 1.2 построена на основе функций API. Даже функция lstrlen, с помощью которой мы получаем длину строки, — это API-функция. А теперь посмотрим в листинге 1.3, как распознает исполняемый код дизассемблер IDA Pro[9].

Листинг 1.3

```
.text:00401000 _main    proc near        ; CODE XREF: start+16E?p
.text:00401000    push  ebx
.text:00401001    mov   ebx, ds:FreeConsole
.text:00401007    push  esi
.text:00401008    push  edi
.text:00401009    call  ebx            ; FreeConsole
.text:0040100B    call  ds:AllocConsole
.text:00401011    mov   edi, ds:GetStdHandle
.text:00401017    push  0FFFFFFF5h     ; nStdHandle
.text:00401019    call  edi            ; GetStdHandle
.text:0040101B    push  0FFFFFFF6h     ; nStdHandle
.text:0040101D    mov   esi, eax
.text:0040101F    call  edi            ; GetStdHandle
.text:00401021    push  0              ; lpReserved
.text:00401023    mov   edi, eax
.text:00401025    mov   eax, lpString
.text:0040102A    push  offset NumberOfCharsWritten
                                       ; lpNumberOfCharsWritten
.text:0040102F    push  eax            ; lpString
.text:00401030    call  ds:lstrlenA
.text:00401036    mov   ecx, lpString
.text:0040103C    push  eax            ; nNumberOfCharsToWrite
.text:0040103D    push  ecx            ; lpBuffer
.text:0040103E    push  esi            ; hConsoleOutput
.text:0040103F    call  ds:WriteConsoleA
.text:00401045    push  0              ; lpReserved
.text:00401047    push  offset NumberOfCharsWritten
                                       ; lpNumberOfCharsRead
.text:0040104C    push  64h            ; nNumberOfCharsToRead
.text:0040104E    push  offset unk_4072C8  ; lpBuffer
.text:00401053    push  edi            ; hConsoleInput
```

[9] Здесь и далее мы будем использовать программу-дизассемблер IDA PRO версии 4.7.

```
.text:00401054    call  ds:ReadConsoleA
.text:0040105A    push  esi           ; hObject
.text:0040105B    mov   esi, ds:CloseHandle
.text:00401061    call  esi           ; CloseHandle
.text:00401063    push  edi           ; hObject
.text:00401064    call  esi           ; CloseHandle
.text:00401066    call  ebx           ; FreeConsole
.text:00401068    pop  edi
.text:00401069    pop  esi
.text:0040106A    xor  eax, eax
.text:0040106C    pop  ebx
.text:0040106D    retn
.text:0040106D _main   endp
```

Даже неопытному программисту ясно, что дизассемблер IDA Pro прекрасно справился с дизассемблированием исполняемого кода. Впрочем, мы пока не намерены разбирать листинги дизассемблеров, времени для этой работы у нас будет достаточно в следующих главах. Я всего лишь хочу акцентировать еще раз ваше, дорогие читатели, внимание на том, что из программ, написанных с помощью лишь API-функций, получается достаточно прозрачный для понимания исполняемый код.

Когда речь идет о программах, подобных представленной в листинге 1.2, мы, однако, вместо того, чтобы использовать API-функции, чаще всего пользуемся библиотечными функциями языка С++. В листинге 1.4 представлена как раз такая программа.

Листинг 1.4

```
#include <stdio.h>
char * s="Example of console program.\n\0";
char buf[100];
void main()
{
  puts(s);
  gets(buf);
}
```

Правда программа из листинга 1.4 не создает собственной консоли, а пользуется тем, что ей предоставляет операционная система. Но в целом все то же самое. Для работы же с консолью используются библиотечные функции

puts и gets. Интересно, что дизассемблер IDA Pro легко справляется и со стандартными библиотечными функциями языка C++. Углубившись в код, например, функции puts, мы достаточно легко можем обнаружить, что выполнение данной функции в конечном итоге сводится к выполнению функции API WriteFile, которая в данном случае эквивалентна функции WriteConsole.

Но ведь при программировании часто используются и нестандартные библиотеки, функции которых не так-то просто "раскрутить" на предмет того, для чего же они предназначены. Так, в частности, происходит, если вы попытаетесь дизассемблировать программу, написанную на Delphi. Например, там для выполнения такого консольного оператора, как write, требуется вызвать две библиотечные процедуры, назначение которых дизассемблер IDA Pro уже не способен распознать.

Итак, линейная структура (или, как мы ее называем, пакетная структура) консольной программы достаточно проста. И хотя сами действия могут быть весьма сложными, их следование друг за другом, несомненно, облегчает исследование кода. Однако если вы захотите писать программу, которая бы более тесно взаимодействовала с пользователем, вам придется обрабатывать события от клавиатуры и мыши. И здесь структура программы станет гораздо сложнее. Вам придется вводить функцию для обработки основных событий консоли и цикл обработки событий от клавиатуры и мыши. В листинге 1.5 представлена примерная схема работы такой программы.

Листинг 1.5

```
#include <windows.h>

BOOL WINAPI handler(DWORD);
void inputcons();
void print(char *);

HANDLE h1,h2;
char * s1 = "Error input!\n";
char s2[35];
char * s4 = "CTRL+C\n";
char * s5 = "CTRL+BREAK\n";
char * s6 = "CLOSE\n";
char * s7 = "LOGOFF\n";
char * s8 = "SHUTDOWN\n";
char * s9 = "CTRL\n";
char * s10="ALT\n";
```

```
char * s11="SHIFT\n";
char * s12=" \n";
char * s13="Code %d \n";
char * s14="CAPSLOCK \n";
char * s15="NUMLOCK \n";
char * s16="SCROLLOCK \n";
char * s17="Enhanced key (virtual code) %d \n";
char * s18="Function key (virtual code) %d \n";
char * s19="Left mouse button\n";
char * s20="Right mouse button\n";
char * s21="Double click\n";
char * s22="Wheel was rolled\n";
char * s23="Character '%c' \n";
char * s24="Location of cursor x=%d y=%d\n";

void main()
{
//инициализация консоли
  FreeConsole();
  AllocConsole();
//получить дескриптор вывода
  h1=GetStdHandle(STD_OUTPUT_HANDLE);
//получить дескриптор ввода
  h2=GetStdHandle(STD_INPUT_HANDLE);
//установить обработчик событий
  SetConsoleCtrlHandler(handler,TRUE);
//вызвать функцию с циклом обработки сообщений
  inputcons();
//удалить обработчик
  SetConsoleCtrlHandler(handler,FALSE);
//закрыть дескрипторы
  CloseHandle(h1); CloseHandle(h2);
//освободить консоль
  FreeConsole();
//выйти из программы
  ExitProcess(0);
};
//обработчик событий
```

```
BOOL WINAPI handler(DWORD ct)
{
//событие CTRL+C?
  if(ct==CTRL_C_EVENT) print(s4);
//событие CTRL+BREAK?
  if(ct==CTRL_BREAK_EVENT) print(s5);
//закрытие консоли?
  if(ct==CTRL_CLOSE_EVENT)
  {
    print(s6);
    Sleep(2000);
    ExitProcess(0);
  };
//завершение сеанса?
  if(ct==CTRL_LOGOFF_EVENT)
  {
    print(s7);
    Sleep(2000);
    ExitProcess(0);
  };
//завершение работы?
  if(ct==CTRL_SHUTDOWN_EVENT)
  {
    print(s8);
    Sleep(2000);
    ExitProcess(0);
  };
  return TRUE;
};
//функция с циклом обработки сообщений консоли
void inputcons()
{
  DWORD n;
  INPUT_RECORD ir;
  while(ReadConsoleInput(h2,&ir,1,&n))
  {
//здесь обработка событий мыши
    if(ir.EventType==MOUSE_EVENT)
    {
```

```
//двойной щелчок
      if(ir.Event.MouseEvent.dwEventFlags==DOUBLE_CLICK)
        print(s21);
//движение мыши по консоли
      if(ir.Event.MouseEvent.dwEventFlags==MOUSE_MOVED)
      {
        wsprintf(s2,s24,ir.Event.MouseEvent.dwMousePosition.X,
          ir.Event.MouseEvent.dwMousePosition.Y);
        print(s2);
      };
//колесико мыши
      if(ir.Event.MouseEvent.dwEventFlags==MOUSE_WHEELED)
        print(s22);
//левая кнопка
if(ir.Event.MouseEvent.dwButtonState==FROM_LEFT_1ST_BUTTON_PRESSED)
    print(s19);
//правая кнопка
      if(ir.Event.MouseEvent.dwButtonState==RIGHTMOST_BUTTON_PRESSED)
        print(s20);
    };
    if(ir.EventType==KEY_EVENT)
    {
      if(ir.Event.KeyEvent.bKeyDown!=1)continue;
//расширенная клавиатура
      if(ir.Event.KeyEvent.dwControlKeyState==ENHANCED_KEY)
      {
        wsprintf(s2,s17,ir.Event.KeyEvent.wVirtualKeyCode);
        print(s2);
      };
//клавиша CAPS LOCK?
      if(ir.Event.KeyEvent.dwControlKeyState==CAPSLOCK_ON)
        print(s14);
//левый ALT?
      if(ir.Event.KeyEvent.dwControlKeyState==LEFT_ALT_PRESSED)
        print(s10);
//правый ALT?
      if(ir.Event.KeyEvent.dwControlKeyState==RIGHT_ALT_PRESSED)
        print(s10);
//левый CTRL?
```

```
      if(ir.Event.KeyEvent.dwControlKeyState==LEFT_CTRL_PRESSED)
        print(s9);
//правый CTRL?
      if(ir.Event.KeyEvent.dwControlKeyState==RIGHT_CTRL_PRESSED)
        print(s9);
//клавиша SHIFT?
      if(ir.Event.KeyEvent.dwControlKeyState==SHIFT_PRESSED)
        print(s11);
//клавиша NUM LOCK
      if(ir.Event.KeyEvent.dwControlKeyState==NUMLOCK_ON)
        print(s15);
//клавиша SCROLL LOCK
      if(ir.Event.KeyEvent.dwControlKeyState==SCROLLLOCK_ON)
        print(s16);
//обработка обычных клавиш
      if(ir.Event.KeyEvent.uChar.AsciiChar>=32)
      {
        wsprintf(s2,s23,ir.Event.KeyEvent.uChar.AsciiChar);
        print(s2);
      }else
      {
        if(ir.Event.KeyEvent.uChar.AsciiChar>0)
        {
//здесь клавиши с кодом больше 0 и меньше 32
          wsprintf(s2,s13,ir.Event.KeyEvent.uChar.AsciiChar);
          print(s2);
        }else
        {
//назовем эти клавиши функциональными
          wsprintf(s2,s18,ir.Event.KeyEvent.wVirtualKeyCode);
          print(s2);
        };
      };
    };
  };
//сообщение об ошибке
  print(s1);
  Sleep(5000);
```

```
};
//функция вывода на консоль
void print(char *s)
{
  DWORD n;
  WriteConsole(h1,s,lstrlen(s),&n,NULL);
};
```

В нашу задачу не входит разбирать данную программу, поскольку я рассчитываю на подготовленного в программировании читателя, но всем интересующимся программированием консольных приложений могу порекомендовать мои книги по программированию в Windows [1—4].

Анализируя программу из листинга 1.5, можно обнаружить примечательную деталь: функция `handler` не вызывается явно. Ее адрес указывается в функции API `SetConsoleCtrlHandler`. Следовательно, единственный способ выйти на эту весьма важную часть программы — это получить ее адрес, анализируя вызов функции `SetConsoleCtrlHandler`. Именно так и поступает дизассемблер IDA PRO. Взгляните на следующий фрагмент

```
.text:00401453    mov    edi, ds:SetConsoleCtrlHandler
.text:00401459    push   1                    ; Add
.text:0040145B    push   offset loc_401000    ; HandlerRoutine
.text:00401460    mov    hConsoleInput, eax
.text:00401465    call   edi                  ; SetConsoleCtrlHandler
```

Посмотрите, дизассемблер не только правильно показывает сам вызов функции `SetConsoleCtrlHandler`, но и совершенно верно трактует оба ее параметра. Пусть читателя не смущает команда

```
mov hConsoleInput, eax
```

Она, разумеется, к вызову функции `ConsoleCtrlHandler` не имеет никакого отношения, а относится к предыдущему вызову функции `GetStdHandle` — издержки оптимизации.

Замечание

Да, надо констатировать, что современные компиляторы часто гораздо лучше умеют оптимизировать код, чем программисты, профессионально пишущие на ассемблере. Программист всегда связан разными условностями, например, хорошей читаемостью программы, а для компилятора это не имеет никакого значения. Впрочем, о некоторых способах оптимизации мы еще поговорим.

Но вернемся опять к приведенному выше фрагменту. Благодаря функции `SetConsoleCtrlHandler` дизассемблер совершенно верно определяет начало функции `handler`, что позволяет ему правильно ее дизассемблировать.

Обратим внимание на функцию `inputcons`. В принципе в ней нет ничего необычного. Циклический вызов функции `ReadConsoleInput` позволяет обнаружить те события, которые не отлавливаются функцией `handler`. Данный цикл можно назвать циклом обработки сообщений консольного приложения. Подобный цикл более характерен для оконных приложений, но, как видите, для консольных приложений это также вполне законный метод программирования. Разумеется, между двумя способами обработки сообщений есть существенная разница. Действительно, приложение может иметь лишь одно консольное окно, поэтому не встает вопрос, к какому окну будет относиться данное сообщение. Приложение GUI может иметь множество окон, а цикл обработки сообщений — один *(см. разд. 1.33)*, и там каждое сообщение маркируется дескриптором окна, к которому данное сообщение обращено. Здесь и возникает определенная сложность, но о ней мы поговорим в следующем разделе.

1.3.3. Оконные приложения

Оконные или графические приложения операционной системы Windows строятся именно на событийных механизмах. Другими словами, большая часть кода таких программ сосредоточена в специальных функциях, которые подобно функции `handler` из предыдущего раздела вызываются системой при наступлении какого-либо события. Кроме этого, для такого типа приложений характерно наличие цикла обработки сообщений, с помощью которого пришедшее в приложение сообщение препровождается соответствующей функции обработки (листинг 1.6).

Листинг 1.6

```
#include <windows.h>

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR     lpCmdLine,
                     int       nCmdShow)
{
  char cname[]="Class";
```

```
char title[]="Простое оконное приложение";
MSG msg;

//структура для регистрации класса окон
WNDCLASS wc;
wc.style          =0;
wc.lpfnWndProc    =(WNDPROC)WndProc;
wc.cbClsExtra     =0;
wc.cbWndExtra     =0;
wc.hInstance      =hInstance;
wc.hIcon          =LoadIcon(hInstance, (LPCTSTR)IDI_APPLICATION);
wc.hCursor        =LoadCursor(NULL, IDC_ARROW);
wc.hbrBackground  =(HBRUSH)(COLOR_WINDOW+1);
wc.lpszMenuName   =0;
wc.lpszClassName  =cname;
//регистрируем класс
if(!RegisterClass(&wc)) return 0;
//создать окно
HWND  hWnd = CreateWindow(
  cname,  //класс
  title,  //заголовок
  WS_OVERLAPPEDWINDOW,  //стиль окна
  0,      //координата X
  0,      //координата Y
  500,    //ширина окна
  300,    //высота окна
  NULL,   //дескриптор окна-родителя
  NULL,   //дескриптор меню
  hInstance,  //идентификатор приложения
  NULL);      //указатель на структуру, посылаемую
              //по сообщению WM_CREATE
//проверим, создалось ли окно
if (!hWnd)    return 0;
//показать окно
ShowWindow(hWnd, nCmdShow);
//обновить содержимое окна
UpdateWindow(hWnd);
```

```
  //цикл обработки сообщений
  while (GetMessage(&msg, NULL, 0, 0))
  {
  //транслировать коды виртуальных клавиш в ASCII-коды
    TranslateMessage(&msg);
  //переправить сообщение процедуре окна
    DispatchMessage(&msg);
  }

  return 0;
};

//процедура окна
LRESULT CALLBACK WndProc(HWND hWnd,
                         UINT message,
                         WPARAM wParam,
                         LPARAM lParam)
{
  switch(message)
  {
  //сообщение при создании окна
  case WM_CREATE:
    break;
  //сообщение при закрытии окна
  case WM_DESTROY:
  //необходимо для  выхода из цикла обработки сообщений
    PostQuitMessage(0);
    break;
  //сообщение, приходящее при перерисовки окна
  case WM_PAINT:
    break;
  //возврат необработанных сообщений
  default:
    return DefWindowProc(hWnd, message, wParam, lParam);
  }
  return 0;
}
```

В листинге 1.6 представлено минимальное оконное приложение, но обладающее всеми основными функциональными особенностями подобных программ. Вообще, оконные приложения строятся на основе главного окна. Все остальное "вращается" вокруг этого окна, как планеты вокруг Солнца. Поэтому легко выделить три обязательные составляющие такого приложения:

- определение и регистрация класса окон, к которому и будет принадлежать главное окно;
- цикл обработки сообщений, основная задача которого — "вылавливать" и перенаправлять нужной оконной функции (не только функции главного окна) сообщения, приходящие на данное приложение;
- функция главного окна, а также возможно функции других окон.

Зная о такой закономерности, мы можем целенаправленно осуществлять поиск этих элементов оконного приложения.

Главной в цикле сообщений является функция API `DispatchMessage`. Она и перенаправляет пришедшее сообщение функции данного окна. Структура сообщения имеет следующий вид:

```
typedef struct {
    HWND hwnd;
    UINT message;
    WPARAM wParam;
    LPARAM lParam;
    DWORD time;
    POINT pt;
} MSG
```

Здесь:

- `hwnd` — дескриптор окна, куда адресовано данное сообщение;
- `message` — код сообщения;
- `wParam` — дополнительная информация, может отсутствовать;
- `lParam` — дополнительная информация, может отсутствовать;
- `time` — указывает время, когда сообщение было послано;
- `pt` — определяет координату курсора мыши в момент послания сообщения. Младшее слово — координата x, старшее — y.

Значение `hwnd` и определяет окно, куда должно быть направлено сообщение. Для каждого же окна, точнее класса окон, определена своя функция обработки сообщений (см. листинг 1.6). Разумеется, система знает это, и сообщение приходит туда, куда и следовало. Но мы-то этого не знаем, а между

тем основная часть кода программы либо сосредоточена в таких функциях, либо вызывается из них. Как же быть? Решить эту проблему (по крайней мере, правильно начать решать) можно, если вспомнить, что все оконные функции регистрируются. Регистрация функций осуществляется вместе с регистрацией класса окон. Посмотрите листинг 1.6, в поле `lpfnWndProc` как раз и заносится адрес функции обработки оконных сообщений. То есть глянув на дизассемблированный код, мы узнаем адрес функции. Вот что, в частности, содержится в листинге дизассемблера IDA Pro:

```
.text:00401077  mov [esp+80h+WndClass.lpfnWndProc], offset loc_401000
```

Здесь `loc_401000` как раз и определяет адрес оконной функции. Программа прекрасно разбирается в функции `RegisterClass` и в той структуре, которая служит аргументом этой функции. А вот фрагмент, полученный с помощью также весьма уважаемого дизассемблера W32Dasm версии 10 (листинг 1.7).

Листинг 1.7

```
:00401077 C744241800104000    mov [esp+18], 00401000
:0040107F 896C241C            mov dword ptr [esp+1C], ebp
:00401083 896C2420            mov dword ptr [esp+20], ebp
:00401087 89742424            mov dword ptr [esp+24], esi

* Reference To: USER32.LoadIconA, Ord:01BDh
                                  |
:0040108B FF15C4504000        Call dword ptr [004050C4]
:00401091 68007F0000          push 00007F00
:00401096 55                  push ebp
:00401097 89442428            mov dword ptr [esp+28], eax

* Reference To: USER32.LoadCursorA, Ord:01B9h
                                  |
:0040109B FF15C8504000        Call dword ptr [004050C8]
:004010A1 89442424            mov dword ptr [esp+24], eax
:004010A5 8D44240C            lea eax, dword ptr [esp+0C]
:004010A9 8D542450            lea edx, dword ptr [esp+50]
:004010AD 50                  push eax
:004010AE C744242C06000000    mov [esp+2C], 00000006
:004010B6 896C2430            mov dword ptr [esp+30], ebp
:004010BA 89542434            mov dword ptr [esp+34], edx
```

```
* Reference To: USER32.RegisterClassA, Ord:0216h
                                          |
:004010BE FF15CC504000          Call dword ptr [004050CC]
:004010C4 6685C0                test ax, ax
```

Рассмотрев листинг дизассемблера W32Dasm, можно сделать вывод, что его анализ гораздо менее информативен, чем у дизассемблера IDA Pro. Все же ему удается в большинстве случаев правильно определить функции API. Так что мы легко находим в начале функцию `RegisterClass`, а затем по другим функциям, предшествующим `RegisterClass`, можем сделать вывод, что команда `mov [esp+18],00401000` и есть присвоение полю `lpfnWndProc` значения адреса функции окна. Итак, узнав функцию окна, мы можем теперь проанализировать текст этой функции и найти нужный фрагмент, выполняющий то или иное действие.

Сама оконная функция предназначена для обработки сообщений, которые на нее приходят. Имеется огромное количество сообщений, извещающих о событиях, происходящих с окном или же элементами, на нем расположенными. Наконец, в функцию окна можно посылать и сообщения, определяемые пользователем. Для этого существует специальная константа `WM_USER`, и все сообщения, определяемые программным путем, должны быть больше или равны этой константе. По тексту оконной функции можно легко определить реакцию на то или иное сообщение, и тем самым понять механизмы работы оконного приложения.

Проблема, однако, в том, что сама оконная функция относится не к конкретному окну, а целому классу окон. Конечно, нередко, особенно это касается того случая, когда приложение строится на основе API-программирования, одному окну соответствует одна функция. Но очень часто это совсем не так. Обработка сообщений для разных окон не составляет большого труда, т. к. в сообщении присутствует дескриптор окна. Но в этом содержится определенная сложность для анализа исполняемого кода, поскольку при статическом анализе кода достаточно сложно определить, для какого окна данный участок обрабатывает сообщение. Здесь на помощь приходят отладчики, с помощью которых можно установить точки останова (breakpoint) на код оконной функции или, как в случае отладчика Soft Ice, на конкретное сообщение конкретного окна и выяснить каким участком кода обрабатываются сообщения конкретного окна.

Разумеется, в оконной программе очень важную роль играет цикл обработки сообщений. Обнаружив его в дизассемблируемом коде, можно выйти на блок программы, предшествующий циклу, т. е. обнаружить, где создается основное окно и регистрируется класс основного окна. Искать цикл обработки сообщений можно по таким функциям API, как `GetMessage`, `PeekMessage`, `TranslateMessage` и `DispatcheMessage`, а также `IsDialogMessage`.

1.3.4. Приложения на основе диалоговых окон

В листинге 1.8 представлен пример, когда главным окном является *модальное диалоговое окно* (рис. 1.6). Модальное диалоговое окно отличается от обычного окна.

- Модальное диалоговое окно создается на основе шаблона, хранящегося в ресурсах или созданного в памяти. В примере из листинга 1.8 диалоговое окно создается на основе шаблона, хранящегося в файле ресурсов.
- Для создания модального диалогового окна используется функция `DialogBoxParam`. Четвертым параметром функции как раз указывается адрес функции обработки сообщений диалогового окна. Функция `DialogBoxParam` не возвращает управление до тех пор, пока не будет вызвана функция `EndDialog`.
- Функция обработки сообщений диалогового окна очень похожа на функцию обработки сообщений обычного окна. Отличия очень незначительные. Если обработку сообщений берет на себя функция, то она возвращает `true`, в противном случае возвращается `false`. Что касается сообщений, то отличие в основном заключается в том, что на диалоговое окно приходит сообщение `WM_INITDIALOG` вместо `WM_CREATE`.
- Как видим, для диалогового окна нет цикла обработки сообщений, как в случае обычного окна. Точнее цикл-то есть, но его создает система, она же и берет на себя обработку и перенаправление сообщений. Так что, повторю, вы можете встретить приложение, в котором отсутствует цикл обработки сообщений.
- Важным моментом работы с модальными диалоговыми окнами является обработка сообщения `WM_CLOSE` при вызове функции `EndDialog`, которая удаляет из памяти модальное диалоговое окно.

Кстати, типичным примером модального диалогового окна является окно, вызываемое функцией API `MessageBox`. Здесь уже система берет на себя не только обработку сообщений, но и создание шаблона окна, и организацию функции сообщений окна.

Листинг 1.8

```
//идентификаторы ресурсов
//определение констант стилей
#define WS_VISIBLE        0x01000000L
#define WS_SYSMENU        0x00080000L
#define WS_MINIMIZEBOX    0x00020000L
#define WS_MAXIMIZEBOX    0x00010000L
```

```
//определение модального диалогового окна
DIALOG DIALOGEX 10, 10, 150, 100
STYLE  WS_VISIBLE | WS_SYSMENU | WS_MINIMIZEBOX |WS_MAXIMIZEBOX
CAPTION "Модальное диалоговое окно"
FONT 12, "Arial"
{
}

//программный модуль
#include <windows.h>

int DWndProc(HWND,UINT,WPARAM,LPARAM);

__stdcall WinMain(HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPSTR lpCmdLine,
    int nCmdShow
)
{
//создать немодальное диалоговое окно
DialogBoxParam(hInstance,"DIALOG",NULL,(DLGPROC)DWndProc,0);
//закрыть приложение
ExitProcess(0);
};
//функция обработки сообщений модального окна
int DWndProc(HWND hwndDlg,UINT uMsg, WPARAM wParam, LPARAM lParam)
  {
  switch(uMsg)
  {
//сообщение, приходящее при создании диалогового окна
  case WM_INITDIALOG:
    break;
//сообщение, приходящее при попытке закрыть окно
  case WM_CLOSE:
    EndDialog(hwndDlg,0);
    return TRUE;
//сообщение от элементов управления
  case WM_COMMAND:
```

```
    break;
  };
    return FALSE;
  };
```

Примечание

В файле ресурсов (см. листинг 1.8) мы явно определяем константы — стили окна. Но это совсем необязательно. Вы можете просто указать строку `#include <windows.h>`. Наконец, можно воспользоваться мастером создания ресурсов Visual Studio .NET и вообще не интересоваться содержимым файла.

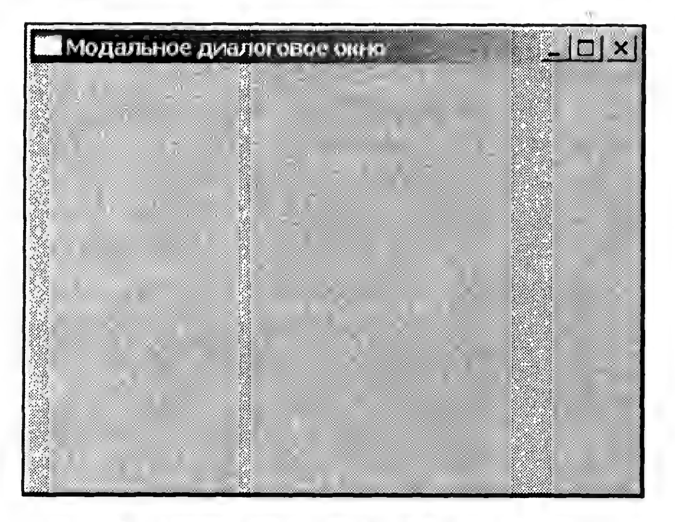


Рис. 1.6. Пример диалогового окна (см. листинг 1.8)

И опять продемонстрирую, как справилась с дизассемблированием программа IDA Pro (листинг 1.9). Функция `DialogBoxParam` помогает найти функцию обработки сообщений диалогового окна.

Листинг 1.9

```
.text:00401000 ; BOOL __stdcall DialogFunc(HWND,UINT,WPARAM,LPARAM)
.text:00401000 DialogFunc  proc near  ; DATA XREF: WinMain(x,x,x,x)+6?
.text:00401000
.text:00401000    hDlg    = dword ptr  4
.text:00401000    arg_4   = dword ptr  8
.text:00401000
```

```
.text:00401000    cmp     [esp+arg_4], 10h
.text:00401005    jnz     short loc_401014
.text:00401007    mov     eax, [esp+hDlg]
.text:0040100B    push    0            ; nResult
.text:0040100D    push    eax          ; hDlg
.text:0040100E    call    ds:EndDialog
.text:00401014
.text:00401014 loc_401014:             ; CODE XREF: DialogFunc+5?j
.text:00401014    xor     eax, eax
.text:00401016    retn
.text:00401016 DialogFunc endp
.text:00401016
.text:00401016 ; ---------------------------------------------------------
.text:00401017               align 10h
.text:00401020
.text:00401020 ; ------- S U B R O U T I N E ----------------------------
.text:00401020
.text:00401020
.text:00401020 ; __stdcall WinMain(x,x,x,x)
.text:00401020 _WinMain@16 proc near             ; CODE XREF: start+186
.text:00401020
.text:00401020 hInstance   = dword ptr  4
.text:00401020
.text:00401020    mov      eax, [esp+hInstance]
.text:00401024    push     0                     ; dwInitParam
.text:00401026    push     offset DialogFunc     ; lpDialogFunc
.text:0040102B    push     0                     ; hWndParent
.text:0040102D    push     offset TemplateName   ; lpTemplateName
.text:00401032    push     eax                   ; hInstance
.text:00401033    call     ds:DialogBoxParamA
; Create a modal dialog box from a
.text:00401033              ; dialog box template resource
.text:00401039    push     0                     ; uExitCode
.text:0040103B    call     ds:ExitProcess
.text:00401041    int      3
; Trap to Debugger
.text:00401041 _WinMain@16 endp
```

Наконец, следует упомянуть еще один тип окна — *немодальные диалоговые окна*. Для этого типа окон необходим явный цикл обработки сообщений. В листинге 1.10 приведен пример построения приложения, где главным окном является немодальное диалоговое окно.

Листинг 1.10

```
//файл ресурсов
//идентификаторы ресурсов
//определение констант стилей
#define WS_VISIBLE          0x010000000L
#define WS_SYSMENU          0x00080000L
#define WS_MINIMIZEBOX      0x00020000L
#define WS_MAXIMIZEBOX      0x00010000L

//определение немодального диалогового окна
DIALOG DIALOGEX 10, 10, 150, 100
STYLE  WS_VISIBLE | WS_SYSMENU | WS_MINIMIZEBOX | WS_MAXIMIZEBOX
CAPTION "Немодальное диалоговое окно"
FONT 12, "Arial"
{
}

//программный модуль
#include <windows.h>
MSG msg;

int DWndProc(HWND,UINT,WPARAM,LPARAM);

__stdcall WinMain(HINSTANCE hInstance,
                  HINSTANCE hPrevInstance,
                  LPSTR lpCmdLine,
                  int nCmdShow)
{
//немодальное диалоговое окно
  HWND hdlg=CreateDialog(hInstance,"DIALOG",NULL,(DLGPROC)DWndProc);
//цикл обработки сообщений
  while (GetMessage(&msg, NULL, 0, 0))
  {
```

```
    IsDialogMessage(hdlg,&msg);
  }

//закрыть приложение
  ExitProcess(0);
};
//функция немодального окна
int DWndProc(HWND hwndDlg,UINT uMsg, WPARAM wParam, LPARAM lParam)
  {
  switch(uMsg)
  {
  //сообщение, приходящее при создании диалогового окна
  case WM_INITDIALOG:
    break;
  //сообщение, приходящее при попытке закрыть окно
  case WM_DESTROY:
    PostQuitMessage(0);
    break;
  case WM_CLOSE:
    DestroyWindow(hwndDlg);
    return TRUE;
  //сообщение от элементов управления
  case WM_COMMAND:
    break;
  };

    return FALSE;
  };
```

Как видно из листинга 1.10, программа по своей структуре напоминает обычное оконное приложение, но есть и некоторые нюансы.

- Отсутствует блок регистрации класса окон, что, конечно, является очевидным.
- Цикл обработки сообщений несколько видоизменился. Вместо обычных функций `TranslateMessage` и `DispatchMessage` взята функция `IsDialogMessage`. Использование последней связано с проблемой нажатия клавиши <Tab> для перехода между элементами управления окна. Для того чтобы в немодальном диалоговом окне все происходило правильно, используется функция `IsDialogMessage`. В общем случае, когда в

приложении могут быть и обычные окна, и диалоговые немодальные окна, цикл обработки сообщений может выглядеть и так:

```
while (GetMessage(&msg, NULL, 0, 0))
{
if(!IsDialogMessage(hw,&msg))
    {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
    }
}
```

Здесь hw — это дескриптор диалогового немодального окна. Впрочем, функция IsDialogMessage может быть использована и для обычного окна.

Замечание

Вообще, кроме указанных выше функций API, цикл обработки сообщений может содержать и другие функции. Там могут присутствовать дополнительные проверки, вызовы процедур и другой код. Иногда очень сложно понять, что перед нами именно цикл обработки сообщений. Однако наличие таких функций, как GetMessage, PeekMessage, DispatchMessage, TanslateMessage, IsDialogMessage, должно насторожить вас и заставить внимательнее отнестись к исследуемому тексту.

- Бросается в глаза также некоторое отличие в обработке события закрытия окна (щелчок по кнопке закрытия в правом верхнем углу). Дело в том, что в случае обычного окна оно действительно закрывается системой и соответственно на функцию окна приходит сообщение WM_DESTROY, которое мы обрабатываем для выхода из цикла обработки сообщения (PostQuitMessage). В случае немодального диалогового окна оно не закрывается автоматически, поэтому мы обрабатываем сообщение WM_CLOSE и закрываем окно с помощью функции DestroyWindow. Никакой особой тайны здесь нет. Все дело в функции DefWindowProc, которая обрабатывает сообщение WM_CLOSE и вызывает функцию DestroyWindow неявно.

1.4. Формат команд микропроцессора Intel

1.4.1. Общие соображения

Рассматривая список команд микропроцессора Pentium, у вас, дорогие читатели, возможно, возник вопрос: как команды хранятся в памяти и чем, ска-

жем, команда MOV EAX,EBX отличается от команды MOV EAX,EDI? Задача данного раздела — показать некоторые закономерности кодирования команд процессора Intel. Возможно, читатели сами увлекутся процессом анализа форматов команд, и это сослужит им добрую службу в деле исследования исполняемого кода.

На рис. 1.7 представлена область памяти, где расположен код программы. Дамп сделан с помощью отладчика OllyDbg.exe, о котором речь еще впереди. Чтобы расшифровать эту последовательность байтов, превратить их в машинные, а точнее, ассемблерные команды, необходимо знать формат этих команд. Вот на этом мы сейчас и остановимся.

Address	Hex dump
00401000	55 8B EC 53 C7 05 D0 86 40 00 32 00 00 00 C7 05
00401010	C4 86 40 00 58 02 00 00 C7 05 C8 86 40 00 BC 02
00401020	00 00 A1 D0 86 40 00 3B 05 C4 86 40 00 8D 1D 53
00401030	10 40 00 89 1D CC 86 40 00 77 05 A1 C4 86 40 00
00401040	3B 05 C8 86 40 00 76 06 FF 25 CC 86 40 00 A1 C8
00401050	86 40 00 A3 C0 86 40 00 A1 C0 86 40 00 50 68 FC
00401060	60 40 00 E8 06 00 00 00 83 C4 08 5B 5D C3 53 56
00401070	57 BE 70 80 40 00 56 E8 0F 02 00 00 8B F8 8D 44
00401080	24 18 50 FF 74 24 18 56 E8 50 03 00 00 56 57 8B
00401090	D8 E8 7D 02 00 00 83 C4 18 5F 5E 8B C3 5B C3 83
004010A0	3D DC 86 40 00 02 74 05 E8 F6 0E 00 00 FF 74 24
004010B0	04 E8 76 0D 00 00 68 FF 00 00 00 FF 15 40 80 40
004010C0	00 59 59 C3 6A 18 68 00 61 40 00 E8 14 17 00 00
004010D0	BF 94 00 00 00 8B C7 E8 64 18 00 00 89 65 E8 8B
004010E0	F4 89 3E 56 FF 15 08 60 40 00 8B 4E 10 89 0D F8
004010F0	86 40 00 8B 46 04 A3 04 87 40 00 8B 56 08 89 15

Рис. 1.7. Дамп кода программы

Прежде всего, замечу, что длина команды может составлять от одного до десяти и более байтов. На рис. 1.8 представлен общий формат команды процессора Intel. Как видим, структура команды может быть достаточно сложной. Но что нас должно, несомненно, утешить — это то, что микропроцессору удается понять код команды и выполнить ее, а, следовательно, и наше намерение разобраться в этой структуре совсем небезнадежно.

Начнем с префиксов. Как видим, префиксы могут быть, а могут и отсутствовать. Логично было бы предположить, что все указанные на рисунке префиксы должны иметь строго заданный код, так чтобы при расшифровке нельзя было перепутать префикс и код команды. Итак, всего существует четыре вида префиксов:

- *префикс команды* может принимать следующие значения:
  - F3H — префикс повторения REPE/REPZ;
  - F2H — префикс повторения REPNE/REPNZ;
  - F0H — префикс блокировки шины LOCK;

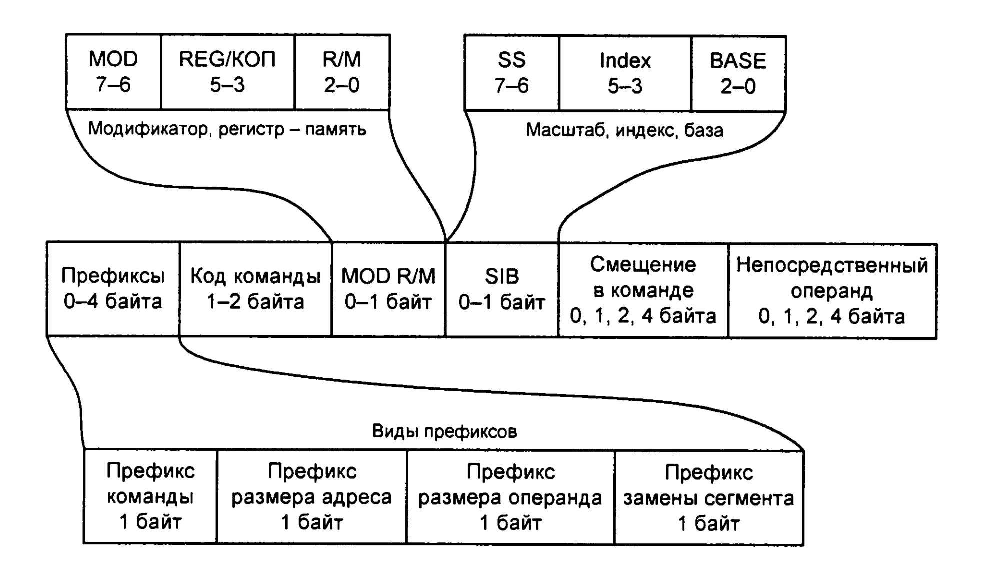


Рис. 1.8. Формат команды микропроцессора Intel

- *префикс размера адреса* (замены размера) принимает значение 67H;
- *префикс размера операнда* (замены размера) принимает значение 66H;
- *префиксы замены сегментов*:
  - для регистра CS — 2EH;
  - для регистра SS — 36H.
  - для регистра DS — 3EH;
  - для регистра ES — 26H;
  - для регистра FS — 64H;
  - для регистра GS — 65H.

Очень важно отметить, что более двух префиксов одного вида не может встречаться в одной команде. Попытка записать такую команду вызовет ошибку процессора.

Итак, зная коды префиксов, мы с достоверностью можем сказать, с чего начинается команда: с кода команды или префикса. Конечно, вы, читатели, понимаете, что это только в том случае, если мы с достоверностью знаем, с какого адреса начинается анализируемая нами команда, в противном случае дизассемблирование начнется с середины команды, и в результате мы получим ассемблерный код, совсем не соответствующий действительности.

1.4.2. Код команды

Обратимся теперь к коду команды процессора. Рассмотрим маленький и очень простой фрагмент программы на ассемблере:

```
PUSH  EAX
PUSH  EBX
PUSH  ECX
POP   ECX
POP   EBX
POP   EAX
RET
```

Как видим, содержимое трех регистров поочередно сохраняется в стеке, а затем значения, хранящиеся в стеке, выталкиваются в те же регистры. Далее происходит выход из текущей процедуры. Так вот, если мы обратимся к области памяти, где хранятся данные команды, то обнаружим следующую последовательность байтов:

```
50 53 51 59 5B 58 C3
```

Первое, что приходит в голову при виде этой последовательности, — на каждую из представленных выше команд приходится один байт. Перед нами, дорогие читатели, типичные представители однобайтовых команд. Таким образом, например, C3H — это не что иное, как код команды RET, точнее RETN. Однако весьма интересны первые шесть битов. Обратимся вначале к командам PUSH. Вот двоичные эквиваленты этих команд: 01010000B (PUSH EAX), 01010011B (PUSH EBX), 01010001B (PUSH ECX). Заметьте, что команды отличаются только младшими битами. Сам собой напрашивается вывод: в команде зашифрована и сама команда, т. е. действие и регистр, подвергающийся данному действию. Чтобы подтвердить нашу догадку, рассмотрим двоичные коды следующих трех команд, т. е. команд POP. Вот они: 01011001B (POP ECX), 01011011B (POP EBX), 01011000B. Ну вот, кажется, ситуация начинает немного проясняться. Сравните, например, двоичные представления команд PUSH EBX и POP EBX. Заметьте, первые два бита совпадают. Но, по сути, совпадают первые два бита и у таких пар, как PUSH EAX и POP EAX и PUSH ECX и POP ECX. А если уж быть совсем точным, то совпадают первые три бита. С другой стороны, у всех команд PUSH совпадают пять последних битов (01010B), соответственно та же ситуация и у всех команд POP (01011B). Закономерность, которую мы обнаружили, не случайна. Действительно, в коде команд PUSH *reg* и POP *reg* зашифрованы не только действия, но и регистры. Коды регистров в действительности универсальны. С этими кодами вы можете столкнуться не только в коде самой команды, но и в байте поля Mod R/M. Но об этом поле речь пойдет несколько позднее.

А сейчас я приведу коды 32-битных рабочих регистров:

- EAX — 000B;
- EBX — 011B;
- ECX — 001B;
- EDX — 010B;
- EDI — 111B;
- ESI — 110B;
- ESP — 100B;
- EBP — 101B.

Казалось бы, все чрезвычайно просто, закономерность нами нащупана, но не тут то было. Во-первых, как быть с 16-битными регистрами, во-вторых, куда девать 8-битные регистры, и в-третьих, и это должно расстроить нас более всего, коды операций PUSH и POP, где операндами выступают не рабочие регистры или же ячейки памяти, совсем другие. Но обо всем по порядку.

Заметим, что команды PUSH и POP невозможны с 8-битными регистрами. Таким образом, проблемы адресации 8-битных регистров в командах PUSH/POP не возникают. А вот как быть с 16-битными регистрами. Вы будете удивлены, но 16-битные регистры кодируются так же, как и 32-битные. То есть, например, регистр AX имеет код 000B, регистр SI имеет код 110B, и т. д. А как же быть с командами пересылки данных в стек, где фигурируют 16-битные регистры? Здесь достаточно просто: перед кодом команды используется префикс замены размера операнда, т. е. 66H. Таким образом, например, команда PUSH AX будет представлена двумя байтами 66 50, а команда POP EAX последовательностью 66 58. Встретив этот префикс, процессор уже знает, что в команде следует заменить операнд с 32-битного на 16-битный. Вывод, который вытекает из полученного нами факта, напрашивается сам собой: использование 32-битных регистров более эффективно, чем 16-битных.

К сожалению, выведенные нами закономерности относительно кодов регистров в командах POP/PUSH на этом и заканчиваются. Вот коды этих команд, примененные к сегментным регистрам:

- PUSH CS — 0EH;
- PUSH DS — 1EH;
- PUSH SS — 16H;
- PUSH ES — 06H;
- PUSH FS — 0FA0H;
- PUSH GS — 0FA8H;

- POP DS — 1FH;
- POP SS — 17H;
- POP ES — 07H;
- POP FS — 0FA1H;
- POP GS — 0FA9H.

Единственное, что можно обнаружить в перечисленных командах, — это то, что коды парных команд (например, PUSH DS/POP DS) отличаются друг от друга на 1. Отметим также, что данные команды для сегментных регистров FS и GS имеют двухбайтовый код. Поскольку данные регистры появились в семействе Intel несколько позднее, для них просто не хватило однобайтовых кодов. Вообще, надо понимать, что разработчики процессора всегда находятся в очень стесненном положении, и требовать от них всеобъемлющих закономерностей не стоит.

Но продолжим наши исследования. Меня, в частности, волнует вопрос: не используются ли байты команд, кроме обозначения самой команды и кода регистра, еще для чего-нибудь?

Обратимся к командам условных переходов. Попытаемся вначале выяснить, каким образом кодируются короткие условные переходы, т. е. переходы в пределах 256 байтов. Рассмотрим небольшой фрагмент на ассемблере, не имеющий никакого практического смысла, но позволяющий нам нащупать некоторые интересные закономерности:

```
        JZ  _LAB
        JNZ _LAB
        JB  _LAB
        JNB _LAB
        JG  _LAB
        JNG _LAB
_LAB:
```

Заглянув в отладчик, мы обнаружим следующую последовательность байтов:

```
74 0A 75 08 72 06 73 04 7F 02 7E 00
```

Ясно, что на каждую команду отводятся два байта, причем второй байт определяет адрес, куда должен быть произведен переход, если выполнится соответствующее условие. Легко увидеть, взглянув на код первой и последней команды, что это просто смещение (см. рис. 1.8) от конца команды. Так что с этим, по крайней мере, на данном этапе, имеется ясность. А вот что собой представляют первые байты команды, то здесь стоит разобраться внимательней. И так, имеем JZ — 01110100B, JNZ — 01110101B, JB — 01110010B, JNB — 01110011B, JG — 01111111B, JNG — 01111110B. Ну что же, вывод ясен: код операции условного перехода — это просто 70H, а четыре младших бита

определяют условие. Причем очевидно, что самый младший бит определяет, как говорят, инвертирование: для JZ бит равен 0, для JNZ — 1 и т. д. Причем соблюдается и некоторая интуитивная логика: "равно нулю" — бит равен нулю, "больше" — бит равен единице. Биты же с 1 по 3 определяют само условие. Поскольку три бита могут задать восемь возможных значений, мы можем на основании табл. 1.10 и наших последних результатов составить табл. 1.26.

Таблица 1.26. *Коды условных переходов*

Команда	**Код**
JB/JNAE/JC	001
JBE/JNA	011
JE/JZ	010
JL/JNGE	110
JLE/JNG	111
JO	000
JP/JPE	101
JS	100

Конечно, у читателя может возникнуть вопрос об условных переходах, где адресом является смещение в 32-битном сегменте. Чтобы исследовать эту проблему, можно несколько видоизменить рассмотренный ранее фрагмент:

```
      JZ  _LAB
      JNZ _LAB
      JB  _LAB
      JNB _LAB
      JG  _LAB
      JNG _LAB
      DB  1000H DUP(0)
_LAB:
```

Вставляя после команды JNB блок данных, мы, таким образом, просто заставляем ассемблер генерировать переходы с 32-битным смещением. Что же получится в результате?

```
0F 84 1E  10 00 00
0F 85 18  10 00 00
0F 82 12  10 00 00
0F 83 0C  10 00 00
```

```
0F 8F 06  10 00 00
0F 8E 00  10 00 00
```

Мы записали результат в виде таблицы, где каждая строка соответствует своей команде. Как видим, код команды теперь состоит из двух байтов. Причем первым байтом везде указано значение 0FH. Структура же второго байта, по сути, уже нам знакома. Код операции 80H, а далее код условия и бит инвертирования. А вот адрес, на первый взгляд, странный какой-то. Господи, как же мы забыли, что адрес (точнее смещение), это, в сущности обычное 32-битное число, и для него должен использоваться стандартный принцип: старший байт в слове имеет старший адрес и старшее слово должно иметь старший адрес. В результате мы получим, что 1E 10 00 00 — это просто 00 00 10 1E, но это как раз в точности расстояние в байтах между командами JNZ _LAB и RET. Та же ситуация получается и для других команд условных переходов.

1.4.3. Байт *MOD R/M*

Рассмотрим простую, на первый взгляд, операцию: MOV EAX,EBX. Код этой операции состоит из двух байтов: 8B C3. Поскольку вариантов пересылок между различными регистрами очень много, то логично было бы считать, что в данном коде зашифрованы оба регистра: EAX и EBX. Разумно также предположить, что это сделано во втором байте, а первый — это код операции. Итак, C3 в двоичном представлении — это 11000011. Чтобы можно было проводить сравнительный анализ, рассмотрим команду MOV EBX,EAX. Код этой команды: 8B D8. Кстати, наше предположение, что первый байт представляет собой код операции, по-видимому, подтверждается. Но D8H — это 11011000B. Сравним данный байт с двоичным представлением C3. Ну, конечно, байты отличаются друг от друга перестановкой троек битов: 000B и 011B. Но это же коды регистров EAX и EBX, о которых мы говорили в предыдущем разделе. Вот здорово, мы почти разгадали код команды MOV, в которой участвуют два 32-битных регистра. Мы с вами столкнулись со структурой байта MOD R/M (см. рис. 1.8), который мы сейчас разберем более подробно.

Итак, байт MOD R/M имеет три следующих поля (см. рис. 1.4):

- поле MOD; данное поле вместе с полем R/M образует 32 возможных значения: 8 регистров и 24 режима индексирования. В приведенном ранее примере поле имело значение 11 и определяло, что поле R/M будет представлять код регистра;
- поле REG/КОП; данное поле обозначает либо код регистра, либо три дополнительные бита кода операции;
- поле R/M; может определять регистр, как местоположение операнда или служить частью кодирования режима адресации совместно с полем MOD.

Возникает законный вопрос, а чем будет отличаться операция MOV EAX,EBX от операции MOV AX,BX? Наверное, вы уже догадались. В последней команде появится впереди (префикс) дополнительный байт — 66H, с которым мы уже сталкивались.

Ну, а как быть с командами MOV, где участвуют 8-битные регистры? Поскольку трех битовых кодов на всех не хватит, то логично предположить, что должен измениться код самой команды. Так в действительности и есть. Например, команда MOV BL,AL будет кодироваться двумя байтами: 8A D8. Заметим, что 8-битных регистров также 8, а, следовательно, они могу быть закодированы с помощью тех же трех битов:

- AL — 000B;
- BL — 011B;
- CL — 001B;
- DL — 010B;
- AH — 100B;
- BH — 111B;
- CH — 101B;
- DH — 110B.

Теперь мы без труда обнаружим в байте D8H регистры BL и AL. Кстати, можно предположить, что команда MOV, в которой участвуют один регистр и один непосредственный операнд, должна обойтись без байта MOD R/M. Действительно, ведь кодировать необходимо всего один операнд. Так и есть, например, код команды MOV EBX,1234H будет равен BB 34120000, а команды MOV ECX,1234H будет B9 34120000. Легко видеть (попробуйте разобраться сами), что кодом команды будет число B8H, а первые три бита станут определять регистр, куда будет помещен непосредственный операнд. Однако вы удивитесь, когда рассмотрите команду MOV EAX,1234H. Код команды будет равен B8 34120000. Разработчики, таким образом, учли, что команда пересылки данных в регистр EAX (аккумулятор) будет производиться чаще, чем в другие регистры, и сделали эту команду короче.

Рассмотрим следующий фрагмент.

```
MOV EAX,DATA1
MOV EBX,DATA1
MOV ECX,DATA1
MOV EDX,DATA1
MOV EDI,DATA1
MOV ESI,DATA1
```

Здесь DATA1 — это некоторая 32-битная переменная. Дизассемблировав фрагмент, получим:

```
A1   00104000
8B1D 00104000
8B0D 00104000
8B15 00104000
8B3D 00104000
8B35 00104000
```

Легко заметить, что и здесь регистр EAX выбивается из общего ряда. Для команды пересылки данных из памяти в регистр имеется собственный код. Что касается остальных команд, то мы видим, что здесь присутствует байт MOD R/M. Переведя шестнадцатеричный код в двоичный, получим, что поле MOD во всех командах равно нулю (00B), поле REG кодирует регистр, а поле R/M равно 101B. Логично предположить, что поля MOD и R/M определяют некоторый режим адресации, одинаковый для представленных команд, кроме той, где используется регистр EAX. Так оно и есть, данный режим предполагает, что эффективный адрес определяется только одним числом — смещением в 32-битном регистре. Кстати, что произойдет, если в вышепредставленных командах поменять местами операнды? Правильно! Изменится только код команды. Все остальное не должно меняться, т. к. не изменился ни способ адресации, ни используемый в команде регистр.

Рассмотрим команду MOV [EBX],ECX. Как видим, в данной команде используется косвенная адресация с помощью регистра EBX. Код подобных операции в такой команде — 89H. Байт же MOD R/M содержит информацию о регистрах и способе адресации — 0B. Легко видеть, что поле MOD содержит 00, а поля REG и R/M — коды регистров ECX и EBX соответственно. Несколько усложним задачу. Рассмотрим команду MOV [EBX+10],ECX. Дизассемблер дает для данной команды последовательность байтов: 89 4B 0A. Как видим, код команды остался тем же. Последний байт, очевидно, является смещением. А вот структура байта MOD R/M имеет следующий вид: 01001011B. В результате, по сравнению с командой MOV [EBX],ECX изменилось только поле MOD, а это и понятно — изменилась адресация. Я думаю, читатель теперь вполне готов принять табл. 1.27, объясняющую использование байта MOD R/M.

Таблица 1.27. Структура байта MOD R/M в 32-битной адресации

Эффективный адрес	Значение поля MOD	Значение поля R/M
[EAX]	00	000
[EBX]	00	011
[ECX]	00	001

Таблица 1.27 (окончание)

Эффективный адрес	Значение поля MOD	Значение поля R/M
[EDX]	00	010
[ESI]	00	110
[EDI]	00	111
Смещ32	00	101
[...]	00	100
Смещ8[EAX]	01	000
Смещ8[EBX]	01	011
Смещ8[ECX]	01	001
Смещ8[EDX]	01	010
Смещ8[ESI]	01	110
Смещ8[EDI]	01	111
Смещ8[EBP]	01	101
Смещ8[...]	01	100
Смещ32[EAX]	10	000
Смещ32[EBX]	10	011
Смещ32[ECX]	10	001
Смещ32[EDX]	10	010
Смещ32[ESI]	10	110
Смещ32[EDI]	10	111
Смещ32[EBP]	10	101
Смещ32[...]	10	100
EAX/AX/AL	11	000
EBX/BX/BL	11	011
ECX/CX/CL	11	001
EDX/DX/DL	11	010
ESP/SP/AH	11	100
EBP/BP/CH	11	101
ESI/SI/DH	11	110
EDI/DI/BH	11	111

Примечание

В табл. 1.27 "Смещ8" означает однобайтовое смещение, "Смещ32" — четырехбайтовое смещение, строка [...] означает, что при данном значении полей MOD и R/M за байтом MOD R/M последует байт SIB.

Взгляните на табл. 1.27. Как видим, байт MOD R/M не позволяет определять такое важное свойство косвенной адресации, как масштабный коэффициент. Для этого служит другой байт SIB. О его наличии можно судить по значению поля R/M, равному 100B.

1.4.4. Байт *SIB*

Наконец мы добрались и до байта SIB. Название этого байта происходит от трех английских слов: scale — масштаб, index — индекс, base — база. Соответственного три поля под таким названием содержат данный байт (см. рис. 1.8):

- биты 7—6, поле Scale, задают масштабный коэффициент;
- биты 5—3, поле Index, задают регистр — индекс;
- биты 2—0 определяют регистр, являющийся базовым (Base).

Рассмотрим следующий фрагмент на языке ассемблера:

```
MOV  [EAX*4][EBX+5],EAX
MOV  [EBX*4][EAX+5],EAX
MOV  [ECX*8][EDX+5],EAX
MOV  [EDX*8][ECX+5],EAX
```

А вот байты, последовательно представляющие данные команды:

```
89 44 83  05
89 44 98  05
89 44 CA  05
89 44 D1  05
```

Очевидно, что код операции во всех случаях равен 89H. Шестнадцатеричное число 44H является не чем иным, как байтом MOD R/M. Представим его в двоичном виде. Итак, 44H = 01000100B. Как видим, поле MOD = 01. Это означает, что в команде должно присутствовать смещение. Так оно и есть, смещение равно 5, и байт, представляющий смещение, идет самым последним. Поле REG равно 000B, что как раз означает, что данные пересылаются из регистра EAX. А вот поле R/M равно 100B и это то самое исключение (см. табл. 1.27), которое говорит нам, что далее должен идти байт SIB. Заметим кстати, что все представленные команды отличаются как раз этим байтом.

Начнем с первой команды. Имеем 83H = 10000011B. Поле Scale, равное 10B, задает масштабный коэффициент, и об этом будет сказано позднее. Поле Index равно 000B, это индексный регистр, и очевидно, что он равен EAX. Регистр EAX действительно используется для формирования результирующего адреса. Поле Base равно 011B и определяет базовый регистр, который, очевидно, равен регистру EBX. По-моему, здесь все ясно, и есть смысл рассмотреть общую картину использования байта SIB (табл. 1.28).

Таблица 1.28. *Структура байта* SIB

Масштабирующий индекс	**Поле Scale**	**Поле Index**
[EAX]	00	000
[EBX]	00	011
[ECX]	00	001
[EDX]	00	010
[EBP]	00	101
[ESI]	00	110
[EDI]	00	111
Не используется	00	100
[EAX*2]	01	000
[EBX*2]	01	011
[ECX*2]	01	001
[EDX*2]	01	010
[EBP*2]	01	101
[ESI*2]	01	110
[EDI*2]	01	111
Не используется	01	100
[EAX*4]	10	000
[EBX*4]	10	011
[ECX*4]	10	001
[EDX*4]	10	010
[EBP*4]	10	101
[ESI*4]	10	110
[EDI*4]	10	111
Не используется	10	100

Таблица 1.28 (окончание)

Масштабирующий индекс	Поле Scale	Поле Index
[EAX*8]	11	000
[EBX*8]	11	011
[ECX*8]	11	001
[EDX*8]	11	010
[EBP*8]	11	101
[ESI*8]	11	110
[EDI*8]	11	111
Не используется	11	100

Теперь, наверное, вам более ясно различие между командами MOV [EAX*8][EBX+10],ECX и MOV [EAX][EBX*8+10],ECX. В первой команде масштабирующий индекс представлен регистром EAX, а во втором случае — регистром EBX, а соответственно с базой — все наоборот. Понятна также чисто техническая невозможность и такой команды: MOV [EAX*4][EBX*2],EAX.

1.4.5. Маленький пример ручного дизассемблирования

Теперь мы, умудренные опытом, можем попытаться дизассемблировать код, представленный на рис. 1.7. Код 55H обозначает команду PUSH EBP. Это легко понять, вспомнив, что код команды PUSH равен 50H, а код регистра EBP равен 5 (101B). Далее идет код 8DH. Ясно, что это не префикс, т. к. коды префиксов нам известны. В принципе за дополнительной информацией можно обратиться к справочнику или набрать команду в отладчике. Оказывается, это код команды LEA. Поскольку сама команда должна иметь два операнда, то, очевидно, что у команды должен быть байт MOD R/M. Следующим у нас идет байт ECH. Представим его в двоичном виде. Имеем ECH = 11101100B. Если все правильно, то первые два бита определяют регистровую адресацию — данные помещаются непосредственно в регистр (см. табл. 1.27). Тогда следующие три бита (REG) определяют регистр, куда будут помещаться данные, а последние биты — регистр, откуда данные будут получены. Этим источником оказывается регистр ESP (код 100B). "Да, — скажете вы, — но ведь команда LEA чаще используется для получения адреса некоторой переменной! Как в этом случае будет выглядеть код команды?" Здесь все в действительности очень просто. Допустим, имеем следующую команду: LEA EBP,DATA1. Результатом дизассемблирования будет последовательность байтов: 8D 2D 00 10 40 00. Ясно, что последние четыре байта —

это просто адрес переменной. А вот что собой представляет байт MOD R/M? Это 2DH = 00101101B. Обратите внимание на последние три бита и посмотрите в табл. 1.27 (при MOD = 00). Это число 101B, которое означает, что эффективным адресом будет смещение в сегменте, т. е. явный адрес переменной. Отсюда понятно, что за вторым байтом следует искать смещение.

Но вернемся к рис. 1.7. Следующий байт равен 53H. И мы легко определяем, что имеем дело с командой PUSH EBX (3 = 011B — это код EBX). Далее имеем C7H. Это код команды MOV, в которой получателем является регистр или ячейка памяти (тип DWORD), а источником — непосредственный операнд. Следующий, очевидно, должен являться байтом MOD R/M. Этот байт равен 05H = 00000101B. Отсюда можно сделать вывод, что непосредственный операнд засылается в ячейку памяти. Далее четыре байта должны представлять адрес ячейки. Вот они: D0 86 40 00, и мы легко определяем, что адрес ячейки это просто 004086D0H. И, наконец, последний байт этой команды — 32H. Таким образом, можно сказать, что нам удалось расшифровать команду MOV DWORD PTR [004086D0H],32. "Почему DWORD?" — спросите вы. Да потому, что команда C7H. Если нужно было использовать команду MOV BYTE PTR [004086D0H],32, то следовало бы использовать код C6H. Итак, вот команды, которые нам удалось расшифровать:

```
PUSH EBP
LEA EBP,ESP
PUSH EBX
MOV DWORD PTR [4086D0H],32
```

Не правда ли занятие утомительное, но после того, что мы узнали в данном разделе — достаточно простое.

Оказывается, некоторые команды микропроцессора могут быть представлены, по крайней мере, двумя различными наборами кодов. Типичный пример: команду MOV EBX,34H транслятор MASM32 транслирует в следующую последовательность кодов: BB 34 00 00 00. При этом код регистра EBX зашифрован в трех первых битах байта кода команды (011B). Но есть и другая возможность закодировать эту же команду с более общих позиций, используя байт MOD R/M. В этом представлении команда будет выражаться следующими байтами: C7 C3 34 00 00 00. Как видим, второе представление команды оказалось на один байт длиннее.

1.4.6. О некоторых проблемах дизассемблирования

Мы очень часто повторяем, что язык ассемблера — это практически то же самое, что машинный язык. Отсюда, казалось бы, можно было сделать вывод, что по машинному коду можно однозначно восстановить ассемблерный

текст программы. Но оказывается, не все так просто. Есть некоторые проблемы, о которых я сейчас расскажу.

Первая проблема касается восстановления структуры данных. Единственная возможность определить структуру данных — проанализировать, как эти данные используются в командах. И вот здесь возникает проблема. Дело в том, что обратиться к данным можно по-разному. Например, легко дизассемблируемая команда типа MOV DWORD PTR [4086D0H],32 показывает, что по адресу 4086D0H расположено некоторое данное (переменная). Это прямая адресация, и здесь все очень просто. А что вы скажете о команде MOV EAX,[EBX]? Чтобы узнать, что находится в регистре EBX, необходим анализ текста программы. Хорошо, если данной команде предшествует, скажем, такая последовательность

```
MOV EAX,4176A0H
ADD EAX,8
MOV EBX,EAX
```

из которой становится ясно, что по адресу 4176A8H расположена некоторая 32-битная переменная. Но очень часто реальный адрес формируется на расстоянии сотен команд от команды, где он используется, посредством довольно сложных манипуляций. В таком случае определить этот адрес можно только при пошаговом выполнении программы, т. е. с помощью механизмов отладки.

Часто бывает недостаточно получить адрес переменной, нужно знать ее размер. Если перед нами, например, массив, то определить, сколько элементов в нем, бывает очень непросто. Даже знание адреса следующей переменной не всегда помогает, ведь между переменными могут быть байты выравнивания.

Данная проблема усугубляется также и наличием двух разных команд, с помощью которых можно получить адрес какого-либо объекта памяти. Традиционно команда LEA была предназначена для получения адреса переменной: LEA EAX,a1. Например, встретив такую последовательность байтов, как 8D 05 08 10 40 00, мы без труда определим, что в регистр EAX засылается адрес, равный 401008H (8DH — код команды LEA, 05H — байт MOD R/M, и подобный анализ мы уже проводили неоднократно). Но в языке ассемблера имеется и еще одна команда: MOV reg32,offset var. Ключевое слово offset заставляет ассемблер подставлять в команду не значение переменной, а ее адрес. Таким образом, понять сразу, без анализа кода, иногда очень серьезного анализа, что это непосредственный операнд или адрес, бывает очень затруднительно.

Другая проблема связана с определением адресов переходов и адресов процедур. Дело в том, что переход на процедуру может осуществляться не только с помощью команды CALL, но и с помощью команды JMP и даже с помо-

щью команды RET. Вот программа, демонстрирующая четыре способа вызова процедуры. Причем три последних способа вызова процедуры могут привести к серьезным затруднениям, которые в определенной ситуации не позволят выяснить, что данный участок представляет собой именно вызываемую откуда-то процедуру.

В листинге 1.11 я привожу программу, в которой используются четыре способа вызова процедуры.

Листинг 1.11

```
.586P
.MODEL FLAT,STDCALL
TEXT SEGMENT
START:
;явный вызов
   CALL PR1
   LEA  EAX,PR1
;косвенный вызов
   CALL EAX
   PUSH OFFSET L1
;адрес возврата в стеке
   JMP  EAX
L1:
   PUSH OFFSET L2
   PUSH EAX
;теперь на вершине стека как раз адрес процедуры,
;а следующим в стеке лежит адрес возврата из процедуры
   RETN ;вызов при помощи RET
L2:
   RETN
PR1 PROC
   RETN
PR1 ENDP
TEXT ENDS
END START
```

И, разумеется, важнейшей проблемой является нахождение правильного адреса, с которого начинается блок команд. В конце концов, если процедуру не удалось идентифицировать при помощи *перекрестных ссылок*, то,

может быть, хотя бы удастся правильно декодировать блок, где располагаются процедуры. Но, увы, и это не всегда просто сделать, во всяком случае, программным путем. Ведь не ясно, где начинается блок процедур. Но предположим, что вам удалось найти первую процедуру, к которой имеется явное обращение. Вы сумели найти и ее конец. Но это, к сожалению, еще не факт, что сразу же за ней расположена следующая процедура. Дело в том, что между процедурами может находиться произвольное количество "пустых" команд, которые может в MASM32, например, вставить директива ALIGN.

Впрочем, о распознавании данных, процедур и других программных структур мы будем подробно говорить в *главе 3*.

1.4.7. О командах арифметического сопроцессора

У читателей, наверное, возникает вопрос об арифметическом сопроцессоре. Есть ли принципиальное отличие в командах сопроцессора от обычных команд процессора Intel Pentium? Сразу отвечу: принципиальных отличий нет. Но есть свои особенности. Минимальная длина команды сопроцессора составляет два байта. Первый байт команды, который ранее мы называли кодом операции, и который таковым уже не является (см. далее), всегда имеет пять старших битов равными 11011B, т. е. старший полубайт первого байта команды сопроцессора всегда равен DH. Это позволяет довольно легко идентифицировать команду сопроцессора среди последовательности байтов памяти.

Кроме первого байта в команде присутствует байт MOD R/M, а также, возможно, операнд, указывающий на память, откуда берется или куда помещается операнд. Рассмотрим команду FLD QWORD PTR [20814000H], помещающую в стек сопроцессора длинное вещественное число из памяти, на которую указывает операнд (адрес). Данная команда представляется следующей последовательностью байтов: DD05 20814000. Первый байт в двоичном представлении имеет следующий вид: 11011101B. Про старшие пять битов мы уже сказали. А вот три младших бита для нас, несомненно, интересны. Данная команда входит в группу команд сопроцессора, манипулирующих операндами, находящимися в памяти. Если младший бит байта равен 1, то команда передает данные в память (из стека сопроцессора) или из памяти. Другие команды имеют бит 0. Это могут быть арифметические операции или операции сравнения. Байты 2 и 1 для рассматриваемых команд определяют тип формата памяти (Memory Format, MF). Имеются четыре возможных значения:

- 00 — короткое вещественное число (32 бита);
- 01 — короткое целое двоичное число (32 бита);

- 10 — длинное вещественное число (64 бита);
- 11 — десятибайтовое число (80 битов).

В нашем случае мы имеем значение 10, т. е. длинное вещественное число. Таким образом, можно констатировать, что первый байт кода команды в сопроцессоре *уже нельзя рассматривать именно как код операции.*

Рассмотрим теперь структуру байта MOD R/M: 05H = 00000101B. Таким образом, MOD = 00B, REG = 000B, R/M = 101B. Глянув на табл. 1.27, мы приходим к вполне очевидному факту, что адрес определяется непосредственным смещением (по значению R/M). Три же средних бита, то, что мы называем REG, в действительности оказались кодом операции (вот она где).

Рассмотрим формат еще одной команды: FADD ST(1),ST(0) (см. табл. 1.21). По этой команде происходит сложение операндов, хранящихся в ST(0) и ST(1), и результат помещается в регистр ST(1). Код команды равен DC C1. В двоичном представлении это будет 11011100 11000001. Рассмотрим вначале первый байт. Нулевой бит равен 0 и в арифметических операциях и операциях сравнения, где участвуют регистры сопроцессора, является частью кода операции. Значение бита за номером 1 определяет, производится ли после операции извлечение из стека (1) или не производится. В данной команде извлечение из стека не производится. Бит за номером 2 показывает, возвращается ли результат в вершину стека (0) или в какой-либо другой регистр (1). В нашем случае результат возвращается в регистр ST(1). Перейдем ко второму байту. Поле MOD равно 11B, и, значит, операция производится с операндами, находящимися в регистрах. Поле R/M равно 001B и определяет второй регистр, участвующий в операции (ST(1)), первым регистром всегда является регистр ST(0). Наконец, код рассмотренной нами операции оказывается состоящим из четырех нулей: 0000B.

Продолжим рассмотрение форматов команд сопроцессора и обратимся к команде FSQRT — извлечение квадратного корня из операнда, находящегося в вершине стека. Для команд, подобных этой (а к ним относятся кроме вычислений трансцендентных функций, загрузка некоторых констант, а также некоторые дополнительные арифметические операции), характерно использование только одного стекового регистра — ST(0). Код данной операции равен D9 FA или в двоичном виде 11011001 11111010. Для такой операции неизменными являются все биты, кроме первых четырех второго байта операции (1010B), которые и определяют, собственно, какая операция производится.

Наконец еще имеется один тип операций, осуществляющих управление сопроцессором. Эти операции не работают ни с какими операндами. Примером такой операции может служить операция FINIT (см. табл. 1.23), которая осуществляет начальную инициализацию сопроцессора. Код этой операции равен DB E3 или в двоичном виде это будет 11011011 11100011. У этих опе-

раций, как и в предыдущем случае, значимыми являются только четыре первых бита второго байта, определяющие, какая операция производится.

На этом мы заканчиваем рассмотрение форматов команд микропроцессора Intel Pentium.

1.5. Описание структуры исполняемого модуля (РЕ-модуль)

Задача данного раздела — познакомить читателей со структурой исполняемого модуля (exe-модуля). Мы исследуем исполняемые модули и, значит, должны знать их структуру. Это важно еще и потому, что, в действительности, такой структурой обладают не только exe-модули, но и динамические библиотеки, объектные модули и драйверы.

1.5.1. Общий подход

Сокращение РЕ расшифровывается как Portable Executable, т. е. дословно *"переносимый исполняемый"*. Данный формат пришел из UNIX, где аналогичный формат называется COFF-форматом (Common Object File Format, стандартный формат объектных файлов). Впрочем, фирмой Microsoft он был значительно переработан и теперь используется ею повсеместно. Как уже было сказано, данный формат применяется не только для обычных исполняемых модулей (exe-модулей), но также и для динамических библиотек (dll-модулей), а также для драйверов режима ядра. Самое интересное заключается в том, что стандарт РЕ распространяется и на объектные файлы (obj-файлы). Наша с вами задача, дорогие читатели, — попытаться охватить РЕ-формат так, чтобы не только понимать его структуру, но и при необходимости использовать свои знания на практике.

Главной особенностью РЕ-модуля является простота его загрузки в память. Не требуется никакой дополнительной настройки. По сути, модуль содержит слепок участка оперативной памяти.

На рис. 1.9 изображена общая схема РЕ-формата исполняемого модуля. Обращает на себя внимание самый первый раздел (на рисунке он изображен сверху). Здесь разработчики отдали дань совместимости с операционной системой MS-DOS. Данный раздел в настоящее время уже не имеет никакого значения. Однако чтобы понять, как это все работает, начнем именно с него. Итак, исполняемый модуль начинается именно с DOS-раздела, который необходим на тот случай, если программу будут запускать в операционной системе MS-DOS. Два первых байта (MZ) — это сигнатура, которая подтверждает, что перед нами исполняемый модуль операционной системы MS-DOS. Сокращение MZ — это инициалы сотрудника фирмы Microsoft

Марка Збиковски (Mark Zbikowscki), разработчика структуры исполняемых модулей в операционной системе MS-DOS. Итак, если запустить PE-программу в операционной системе MS-DOS, то загрузчик этой системы по сигнатуре будет считать, что перед ним обычная программа MS-DOS, и запустит ее обычным способом. В сущности, так и есть: после сигнатуры MZ в PE-модуле идет стандартный заголовок MS-DOS, а далее — маленькая программа, заглушка, которая обычно выводит на текстовый экран сообщение, что данная программа не может выполняться в MS-DOS ("This program cannot be run in DOS mode"), и заканчивает свою работу. Стандартная программа-заглушка, ее называют *stub*, приведена в листинге 1.12.

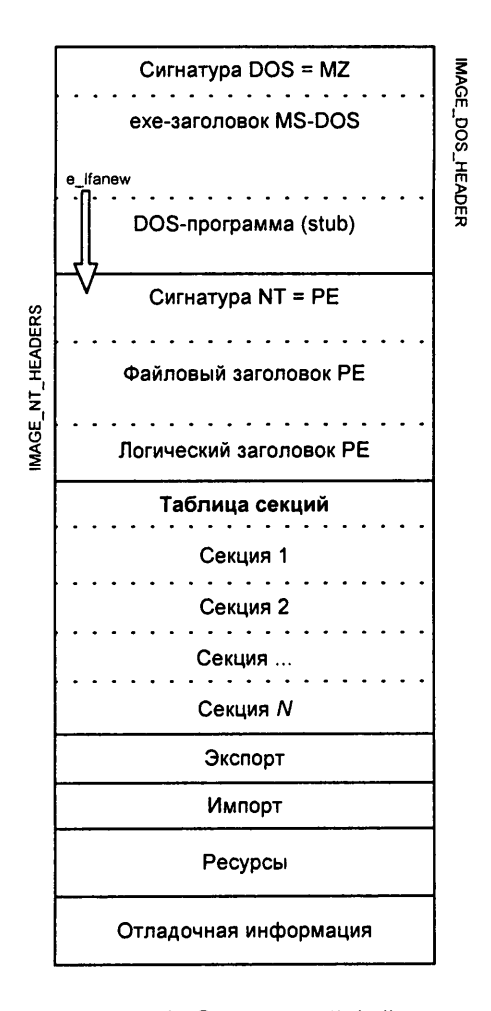


Рис. 1.9. Структура PE-файла

Листинг 1.12

```
PUSH CS
;регистр данных совпадает с регистром кода
POP DS
MOV DX,OFFSET MSG
MOV AH,9
;вывод текстовой строки MSG
INT 21H
MOV AX,4C01H
;выход из программы с кодом 1
INT 21H
MSG DB 'This program cannot be run in DOS mode $'
```

Конечно, программа может быть и другой[10], но какая нам разница, если чистой MS-DOS уже и не найти и, следовательно, заглушка уже никогда не получит управление? Удобнее всего разбирать MZ-заголовок, обратившись к структуре, которую можно найти в заголовочном файле WINNT.H[11]. Вот эта структура (листинг 1.13).

Листинг 1.13

```
struct IMAGE_DOS_HEADER {      // DOS .EXE header
       WORD   e_magic;         // Magic number
       WORD   e_cblp;          // Bytes on last page of file
       WORD   e_cp;            // Pages in file
       WORD   e_crlc;          // Relocations
       WORD   e_cparhdr;       // Size of header in paragraphs
       WORD   e_minalloc;      // Minimum extra paragraphs needed
       WORD   e_maxalloc;      // Maximum extra paragraphs needed
       WORD   e_ss;            // Initial (relative) SS value
       WORD   e_sp;            // Initial SP value
       WORD   e_csum;          // Checksum
       WORD   e_ip;            // Initial IP value
       WORD   e_cs;            // Initial (relative) CS value
```

[10] Был период, когда создатели компьютерных вирусов использовали программу-заглушку для активизации вредоносного кода.

[11] Все структуры, используемые в PE-заголовке, мы будем брать именно из заголовочных файлов.

```
        WORD    e_lfarlc;       // File address of relocation table
        WORD    e_ovno;         // Overlay number
        WORD    e_res[4];       // Reserved words
        WORD    e_oemid;        // OEM identifier (for e_oeminfo)
        WORD    e_oeminfo;      // OEM information; e_oemid specific
        WORD    e_res2[10];     // Reserved words
        LONG    e_lfanew;       // File address of new exe header
  }
```

Нас не будут интересовать поля данной структуры, кроме трех. Собственно с полем e_magic мы уже знакомы. Это просто сигнатура MZ. Поле e_lfarlc (смещение 18H от начала файла) изначально было предназначено, чтобы хранить адрес таблицы размещения. Таблица размещения использовалась загрузчиком MS-DOS для того, чтобы настроить относительные адреса, используемые в программе. Но вот, оказывается, если это поле содержит байт 40H, то это означает, что перед нами как раз PE-модуль[12]. Впрочем, судя по всему, Windows не проверяет содержимое этого поля, а раз так, считать, что значение 40H является достоверным признаком того, что перед нами PE-модуль, наверное, все-таки, не стоит. Наконец, поле e_lfanew содержит относительный адрес (смещение относительно начала файла), откуда начинается PE-заголовок (см. рис. 1.9). По этому адресу должна находиться уже сигнатура PE-модуля, соответственно буквы P и E.

В листинге 1.14 представлена простая программа, с помощью которой можно определить, является данный файл загружаемым PE-модулем или нет. Имя проверяемого модуля следует указать в командной строке.

Со структурой IMAGE_DOS_HEADER мы уже знакомы, структуру IMAGE_NT_HEADERS, которая представляет PE-заголовок, мы рассмотрим в следующих разделах. Данная структура определена в заголовочном файле windows.h, соответственно константы IMAGE_DOS_SIGNATURE и IMAGE_NT_SIGNATURE, определяющие сигнатуры MZ (5A4Dh) и PE (4550h), также содержатся в заголовочном файле.

Листинг 1.14

```
#include <windows.h>
#include <stdio.h>
HANDLE openf(char * );
HANDLE hf;
```

[12] Или NE-модуль, который запускался в Windows 3.1. Эти программы вы теперь редко встретите.

```
IMAGE_DOS_HEADER id;
IMAGE_NT_HEADERS iw;
//главная функция
int main(int argc, char* argv[])
{
  DWORD n;
  int er=0;
  LARGE_INTEGER l;
//проверка наличия параметров
  if(argc<2){printf("No parameters!\n");er=1; goto _exit;};
//первый в списке - имя файла
  if((hf=openf(argv[1]))==INVALID_HANDLE_VALUE)
  {
    printf("No file!\n");
    er=2;
    goto _exit;};
//определим длину файла
  GetFileSizeEx(hf,&l);
//прочитать заголовок DOS
  if(!ReadFile(hf,&id,sizeof(id),&n,NULL))
  {
    printf("Read DOS_HEADER error 1!\n");
    er=3;
    goto _exit;};
  if(n<sizeof(id))
  {
    printf("Read DOS_HEADER error 2!\n");
    er=4;
    goto _exit;};
//проверить сигнатуру DOS ('MZ')
  if(id.e_magic!=IMAGE_DOS_SIGNATURE)
  {
    printf("No DOS signature!\n");
    er=5;
    goto _exit;}
  printf("DOS signature is OK!\n");
  if(id.e_lfanew>l.QuadPart)
  {
    printf("No NT signature!\n");
```

```
    er=6;
    goto _exit;};
//вначале передвинем указатель
  SetFilePointer(hf,id.e_lfanew,NULL,FILE_BEGIN);
//прочитать заголовок NT
  if(!ReadFile(hf,&iw,sizeof(iw),&n,NULL))
  {
    printf("Read NT_HEADER error 1!\n");
    er=7;
    goto _exit;};
  if(n<sizeof(iw))
  {
    printf("Read NT_HEADER error 2!\n");
    er=8;
    goto _exit;};
//проверить сигнатуру NT ('PE')
  if(iw.Signature!=IMAGE_NT_SIGNATURE)
  {
    printf("No NT signature!\n");
    er=9;
    goto _exit;}
  printf("NT signature is OK!\n");
//закрыть дескриптор файла
_exit:
  CloseHandle(hf);
  return er;
};
//функция открывает файл для чтения
HANDLE openf(char * nf)
{
  return CreateFile(nf,
                    GENERIC_READ,
                    FILE_SHARE_WRITE | FILE_SHARE_READ,
                    NULL,
                    OPEN_EXISTING,
                    NULL,
                    NULL);
};
```

Программа из листинга 1.14 не может, разумеется, со 100-процентной гарантией определить, что перед нами правильный PE-модуль. Для этого потребовался бы более детальный анализ PE-заголовка.

В *приложении* представлена программа, которая осуществляет более детальный анализ PE-заголовка. Она является развитием программы из листинга 1.14. В частности, кроме анализа непосредственно самих заголовков, там выводится содержимое секций импорта, экспорта и секции ресурсов.

1.5.2. Заголовок PE

Обратимся теперь к PE-заголовку. Как мы уже выясняли, данный заголовок представляется в виде структуры `IMAGE_NT_HEADERS` (листинг 1.15).

Листинг 1.15

```
struct IMAGE_NT_HEADERS {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;
}
```

Мы видим, что структура состоит из двух частей — `IMAGE_FILE_HEADER` и `IMAGE_OPTIONAL_HEADER32`, не считая поля (сигнатуры) `Signature`, равного PE. Начнем разбирать часть заголовка — `IMAGE_FILE_HEADER`, которую еще называют основным заголовком (листинг 1.16).

Листинг 1.16

```
struct IMAGE_FILE_HEADER {
       WORD    Machine;
       WORD    NumberOfSections;
       DWORD   TimeDateStamp;
       DWORD   PointerToSymbolTable;
       DWORD   NumberOfSymbols;
       WORD    SizeOfOptionalHeader;
       WORD    Characteristics;
}
```

Охарактеризуем кратко поля структуры:

- Machine — тип процессора; для процессоров Intel i80x86 это значение 014ch;
- NumberOfSections — количество секций в PE-модуле;
- TimeDateStamp — дата и время создания файла;
- PointerToSymbolTable — используется при отладке; обычно равно 0;
- NumberOfSymbols — используется при отладке; обычно равно 0;
- SizeOfOptionalHeader — размер второй части заголовка[13] PE (см. далее описание IMAGE_OPTIONAL_HEADER32); Обычно составляет 224 байта;
- Characteristics — здесь содержатся информационные биты (флаги). В частности, 13-й бит определяет, является данный модуль dll-библиотекой (0) или exe-модулем (1).

Перейдем теперь ко второй части заголовка PE — дополнительному заголовку (IMAGE_OPTIONAL_HEADER32). Поля дополнительного заголовка представлены в листинге 1.17.

Листинг 1.17

```
struct IMAGE_OPTIONAL_HEADER {
       WORD    Magic;
       BYTE    MajorLinkerVersion;
       BYTE    MinorLinkerVersion;
       DWORD   SizeOfCode;
       DWORD   SizeOfInitializedData;
       DWORD   SizeOfUninitializedData;
       DWORD   AddressOfEntryPoint;
       DWORD   BaseOfCode;
       DWORD   BaseOfData;
       DWORD   ImageBase;
       DWORD   SectionAlignment;
       DWORD   FileAlignment;
       WORD    MajorOperatingSystemVersion;
       WORD    MinorOperatingSystemVersion;
       WORD    MajorImageVersion;
       WORD    MinorImageVersion;
       WORD    MajorSubsystemVersion;
```

[13] Душа как-то не принимает термин "оптиональный заголовок".

```
        WORD    MinorSubsystemVersion;
        DWORD   Win32VersionValue;
        DWORD   SizeOfImage;
        DWORD   SizeOfHeaders;
        DWORD   CheckSum;
        WORD    Subsystem;
        WORD    DllCharacteristics;
        DWORD   SizeOfStackReserve;
        DWORD   SizeOfStackCommit;
        DWORD   SizeOfHeapReserve;
        DWORD   SizeOfHeapCommit;
        DWORD   LoaderFlags;
        DWORD   NumberOfRvaAndSizes;
IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
}
```

Дадим пояснения к полям:

- `Magic` — определяет основное предназначение данного модуля; в частности, для обычного исполняемого файла это поле равно `010BH`;
- `MajorLinkerVersion` — старший номер версии компоновщика, создавшего данный файл;
- `MinorLinkerVersion` — младший номер версии компоновщика, создавшего данный файл;
- `SizeOfCode` — размер в байтах исполняемого кода, содержащегося в файле;
- `SizeOfInitializedData` — размер секции инициализированных данных;
- `SizeOfUninitializedData` — размер секции неинициализированных данных;
- `AddressOfEntryPoint` — ***относительный виртуальный адрес*** инструкции, с которой начинается выполнение программы. Адрес в виртуальном адресном пространстве относительно адреса загрузки исполняемого модуля будем называть относительным виртуальным адресом (Relative Virtual Address, RVA). Соответственно, если относительный адрес, с которого начнет выполняться модуль, будет `1000H`, а модуль станет загружаться по адресу `400000H` (см. поле `ImageBase`), то точка, откуда начнется выполнение, будет находиться по адресу `401000H`;
- `BaseOfCode` — относительный виртуальный адрес первой программной секции;
- `BaseOfData` — относительный виртуальный адрес, с которого начинается первая секция данных; обычно секции данных начинаются сразу за секциями исполняемого кода;

- `ImageBase` — виртуальный адрес (не относительный), с которого будет загружен модуль. Если загрузчик будет располагать модуль в памяти, начиная именно с этого адреса, то ему не потребуется производить дополнительную настройку адресов, и процесс загрузки будет происходить быстро. Если загрузчику не удастся загрузить модуль по данному адресу, то ему придется производить дополнительную настройку адресов. Для exe-модулей это значение обычно равно `400000H`;
- `SectionAlignment` — значение, определяющее выравнивание секций в памяти; все секции в памяти должны начинаться с адреса, кратного данной величине;
- `FileAlignment` — значение, определяющее выравнивание секций в файле; все секции в файле должны начинаться с адреса, кратного данной величине;
- `MajorOperatingSystemVersion` — старший номер подсистемы Win32, необходимый для запуска программы;
- `MinorOperatingSystemVersion` — младший номер подсистемы Win32, необходимый для запуска программы;
- `MajorImageVersion` — пользовательский номер версии, задаваемый при компоновке (старшая часть `n`); для link.exe ключ имеет вид `/version:n.m`;
- `MinorImageVersion` — пользовательский номер версии, задаваемый при компоновке (младшая часть `m`);
- `MajorSubsystemVersion`, `MinorSubsystemVersion` — старшие и младшие номера версий подсистемы; скорее, все поля никак не используются;
- `Win32VersionValue` — хотя название этого поля вполне осмысленное, во многих статьях по вопросам PE-заголовков указывается, что его значение должно быть 0;
- `SizeOfImage` — общий размер образа PE (заголовки и секции) в памяти, выровненный по `SectionAlignment`;
- `SizeOfHeaders` — размер всех заголовков плюс размер таблицы секций;
- `CheckSum` — контрольная сумма файла; для exe-модуля значение равно 0;
- `Subsystem` — указывает, для какой из подсистем предназначен данный модуль. `0000H` — неизвестная подсистема, `0001H` — драйвер устройства, `0002H` — Windows GUI, `0003H` — консольное приложение, `0005H` — OS/2, `0007H` — POSIX;
- `DllCharacteristics` — начиная с Windows NT 3.5, данное поле перестало использоваться;
- `SizeOfStackReserve` — необходимый объем памяти для стека;
- `SizeOfStackCommit` — выделяемый объем памяти для стека;

- `SizeOfHeapReserve` — необходимый объем памяти для локальной кучи;
- `SizeOfHeapCommit` — выделяемый объем памяти для локальной кучи;
- `LoaderFlags` — начиная с Windows NT 3.5, данное поле перестало использоваться;
- `NumberOfRvaAndSizes` — данное поле зарезервировано для будущего расширения формата (размер массива, содержащего некоторые структуры); обычно равно `10H`;
- `DataDirectory` — массив структур (листинг 1.18). Пока значение `IMAGE_NUMBEROF_DIRECTORY_ENTRIES` равно 16. Каждая структура состоит из двух элементов по 4 байта. Реально работают первые 12 структур. Первый элемент структуры описывает положение данных (относительный виртуальный адрес), второй элемент — размер данных. Вот предназначение элементов массива:
  - 0 — таблица экспортируемых функций;
  - 1 — таблица импортируемых функций;
  - 2 — таблица ресурсов;
  - 3 — таблица исключений;
  - 4 — таблица безопасности;
  - 5 — таблица настройки;
  - 6 — таблица отладки;
  - 7 — строки описания;
  - 8 — характеристика скорости компьютера, измеряемая в MIPS;
  - 9 — область TLS (Thread Local Storage, локальная память цепочки);
  - 10 — область таблицы конфигурации;
  - 11 — таблица адресов импорта.

Листинг 1.18

```
struct IMAGE_DATA_DIRECTORY {
  DWORD   VirtualAddress;
  DWORD   Size;
}
```

1.5.3. Секции

Сразу после дополнительного заголовка PE располагается таблица секций. Для верности можно сравнить значение поля `SizeOfOptionalHeader` (см.

структуру IMAGE_FILE_HEADER) с величиной sizeof(IMAGE_NT_HEADERS)-sizeof(IMAGE_FILE_HEADER)-4. После этого спокойно обратиться к следующему адресу от начала файла e_lfanew+sizeof(IMAGE_NT_HEADERS).

Таблица секций состоит из структур по 40 байтов каждая. Количество же секций берется из поля NumberOfSections (см. структуру IMAGE_FILE_HEADER из листинга 1.16), так что получить список секций не составляет никакого труда. В листинге 1.19 представлена структура, из которых состоит таблица секций.

Листинг 1.19

```
struct IMAGE_SECTION_HEADER {
    BYTE     Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
            DWORD   PhysicalAddress;
            DWORD   VirtualSize;
    } Misc;
    DWORD   VirtualAddress;
    DWORD   SizeOfRawData;
    DWORD   PointerToRawData;
    DWORD   PointerToRelocations;
    DWORD   PointerToLinenumbers;
    WORD    NumberOfRelocations;
    WORD    NumberOfLinenumbers;
    DWORD   Characteristics;
}
```

Разберем поля данной структуры:

- Name — имя секции. Значение IMAGE_SIZEOF_SHORT_NAME равно 8. Если число символов в имени меньше 8, то оставшиеся байты заполняются нулевыми значениями;
- VirtualSize — требуемый для секции размер памяти;
- VirtualAddress — относительный виртуальный адрес, по которому загрузчик должен загрузить секцию;
- SizeOfRawData — размер секции, выровненный в бóльшую сторону согласно значению поля FileAlignment (см. структуру IMAGE_OPTIONAL_HEADER, листинг 1.17);
- PointerToRawData — смещение в файле, по которому находится данная секция;

- `PointerToRelocations`, `PointerToLinenumbers`, `NumberOfRelocations`, `NumberOfLinenumbers` — данные поля используются в obj-файлах и нами рассматриваться не будут;
- `Characteristics` — флаги, характеризующие секцию (табл. 1.29).

***Таблица 1.29.** Флаги, характеризующие секцию*

Значение	Объяснение
00000020H	Секция содержит программный код
00000040H	Секция содержит инициализированные данные
00000080H	Секция содержит неинициализированные данные
00000200H	Секция используется компилятором
00000800H	Секция используется компилятором
04000000H	Секция не может кэшироваться
08000000H	Секция не имеет страничной организации
10000000H	Совместно используемая секция
20000000H	Секция является исполняемой
40000000H	Секция только для чтения
80000000H	Секция может использоваться для записи

Имена секций и их назначение могут быть разными. Это уже на усмотрение компиляторов.

Замечание

Вы сами можете создавать собственные секции и давать им уникальные имена. Например, вы можете написать программу на ассемблере и дать секции (сегменту), где будет располагаться исполняемый код, произвольное имя. Программа будет работать нормально, но некоторые дизассемблеры и отладчики будут возмущены тем, что точка входа в программу расположена в секции с именем, которое им совсем незнакомо.

Вот далеко не полный список секций, создаваемых компиляторами фирмы Microsoft и Borland:

- .text — секция содержит исполняемый код (Microsoft);
- CODE — секция содержит исполняемый код (Borland);
- .data — здесь содержатся инициализированные глобальные переменные (Microsoft);
- DATA — здесь содержатся инициализированные глобальные переменные (Borland);

- .bss — все неинициализированные данные; размер секции в файле равен нулю;
- .CRT — еще одна секция для инициализированных данных (Microsoft);
- CRT — данные (Borland);
- .rdata — данные, доступные только для чтения (константы, отладочная информация);
- .rsrc — секция содержит информацию о ресурсах;
- .edata — секция содержит информацию об экспортируемых функциях;
- .idata — секция содержит информацию об импортируемых функциях;
- .reloc — таблица настроек. Информация может понадобиться загрузчику Windows, если по каким-либо причинам он будет загружать модуль по адресу, отличному от указанного в заголовке PE. Таблица содержит относительные адреса тех ячеек памяти, содержащих, в свою очередь, используемые в программе адреса, значения которых, возможно, потребуется изменить при загрузке. Данная таблица называется также *таблицей перемещения* (relocation table). Об исследовании таблицы перемещений *см. в разд. 2.1.1* (о программе dumpbin.exe);
- .icode — переходы на функции импорта старых версий tlink32.exe;
- .debug — секция содержит отладочную информацию.

Итак, по таблице секций вы можете вычислить положение секции в файле и ее размер и, тем самым, получить возможность посмотреть, что в ней хранится: получить листинг или даже попытаться дизассемблировать исполняемый код. Но особо следует остановиться на таблицах импорта, экспорта и секции, содержащей ресурсы, чем мы займемся в следующих разделах. Вначале, однако, следует прояснить, что такое PE-образ в виртуальной памяти. Дело в том, что это не совсем копия PE-модуля. В упрощенном варианте загрузка модуля происходит в два приема:

1. В виртуальную память загружаются все заголовки: DOS-заголовок, PE-заголовок (`IMAGE_NT_HEADERS`) и таблица секций.
2. В память начинают загружаться секции, причем их относительные виртуальные адреса должны быть выровнены согласно полю `SectionAlignment` (см. описание структуры `IMAGE_OPTIONAL_HEADER`).

Какой вывод можно сделать из этого? Прежде всего, следует понять, как по виртуальному адресу некоторого объекта можно определить его смещение в файле. Этот важный вопрос возникает в связи с таблицами импорта и экспорта. Алгоритм получения смещения таков:

1. По виртуальному адресу определяем секцию, где располагается данный объект.
2. Из таблицы секций определяем смещение секции в файле PE.

3. Определяем смещение объекта внутри секции.
4. Смещение объекта получаем суммированием смещения секции в файле и смещения объекта внутри секции.

В листинге 1.20 представлена функция на C++, определяющая смещение в файле PE по относительному виртуальному адресу. Соответственно предполагается, что заранее прочитана глобальная структура iw=IMAGE_NT_HEADERS и заполнен глобальный массив ais, состоящий из структур IMAGE_SECTION_HEADER (см. листинг 1.19). Параметр vsm на входе — это относительный виртуальный адрес объекта. Функция возвращает смещение в файле PE.

Листинг 1.20

```
//определение смещения в файле PE по относительному виртуальному адресу
DWORD getoffs(DWORD vsm)
{
  DWORD fi=0;
  if(vsm<ais[0].VirtualAddress)return fi;
  for(int i=0; i<iw.FileHeader.NumberOfSections; i++)
  {
    if(vsm<ais[i].VirtualAddress&&i>0){
    fi=ais[i-1].PointerToRawData+(vsm-ais[i-1].VirtualAddress);
      break;};
  };
  if(i==iw.FileHeader.NumberOfSections)
    fi=ais[i-1].PointerToRawData+(vsm-ais[i-1].VirtualAddress);
  return fi;
};
```

1.5.4. Таблица импорта

Сразу хочу сказать главное. Если кто-то вздумает выйти на секцию импорта посредством имени .idata в таблице секций, то его ждет разочарование. По крайней мере, редакторы связей фирмы Microsoft такой секции не создают. Значит, придется использовать массив DataDirectory из структуры IMAGE_OPTIONAL_HEADER (см. листинг 1.17). Вы можете использовать программу из *приложения*, чтобы провести простейшее исследование исполняемых модулей. Вы легко обнаружите, что во многих исполняемых модулях секции .idata нет, хотя таблица импорта присутствует. Если же секция .idata присутствует, то таблица импорта, разумеется, расположена именно там.

Итак, напоминаю. Массив DataDirectory состоит из двенадцати значимых элементов (всего 16). Каждый элемент массива состоит из двух полей: VirtualAddress — виртуальный адрес объекта, Size — размер объекта (см. листинг 1.18). *Таблицу импорта* определяет второй элемент (индекс 1). Это единственное надежное свидетельство о том, где находится таблица импорта. Но нам этого вполне достаточно. Вспомните наши рассуждения в конце предыдущего раздела и листинг 1.20. Таким образом, проблем с поиском таблицы импорта у нас не возникает и остается только понять структуру этой таблицы, чем мы сейчас и займемся.

Сразу же в начале таблицы импорта мы натыкаемся на массив структур, представленных на листинге 1.21.

Листинг 1.21

```
struct IMAGE_IMPORT_DESCRIPTOR {
union {
        DWORD   Characteristics;
        DWORD   OriginalFirstThunk;
    };
        DWORD   TimeDateStamp;
        DWORD   ForwarderChain;
        DWORD   Name;
        DWORD   FirstThunk;
}
```

Массив заканчивается элементом с нулевыми полями. Лишний раз подчеркну, что проверять на равенство нулю следует, по крайней мере, два поля, например, Characteristics и Name. Разберем теперь поля, представленные в листинге 1.21:

- Characteristics — относительный виртуальный адрес другого массива, содержащего относительные виртуальные адреса имен импортируемых функций;
- TimeDateStamp — дата и время создания файла (библиотеки dll) или 0;
- ForwarderChain — обычное значение 0FFFFFFFFh;
- Name — адрес строки в формате ASCII, содержащей имя библиотеки импорта (динамической библиотеки); таким образом, каждый элемент массива соответствует своей динамической библиотеке;
- FirstThunk — относительный виртуальный адрес массива, содержащего адреса имен импортируемых функций, вторая копия массива, на который указывает поле Characteristics. Если поле Characteristics равно 0

(так поступают некоторые компиляторы фирм, отличных от Microsoft), то следует проверить поле `FirstThunk`, которое указывает на вторую копию массива.

Замечание

Я надеюсь, что читатель понимает, что речь идет о динамических библиотеках, неявно связываемых с исполняемым модулем, а не тех, которые загружаются по ходу выполнения программы API-функцией `LoadLibrary`.

Теперь обратимся к массивам, на которые указывают поля `Characteristics` и `FirstThunk`. Лишний раз хочу подчеркнуть, что речь идет о двух массивах, хотя их элементы указывают на одни и те же имена импортируемых функций. Массивы состоят из структур, представленных в листинге 1.22.

Листинг 1.22

```
struct IMAGE_THUNK_DATA32 {
union {
      DWORD ForwarderString;
      DWORD Function;
      DWORD Ordinal;
      DWORD AddressOfData;
      } u1;
}
```

Как видим, структура `IMAGE_THUNK_DATA32`, по сути, состоит из одного поля, но в четырех ипостасях. Это поле указывает (относительный виртуальный адрес) на имя импортируемой функции для данной динамической библиотеки (не потеряли ниточку рассуждений?). Если старшее слово поля равно `8000H`, то младшее слово содержит порядковый номер импортируемой функции (импорт по ординалу). В конце массива должно быть двойное слово, равное нулю.

Ну и, наконец, мы подходим к структуре имени импортируемой функции. Не вдаваясь в подробности, заметим, что имя функции имеет простую структуру ASCII с нулем на конце. Только вот начинается оно по адресу, который указан в структуре `IMAGE_THUNK_DATA32`, плюс два байта. В двух же предшествующих байтах содержится номер из dll-библиотеки для данной импортируемой функции.

Особо следует остановиться на массиве, на который указывает поле `FirstThunk` из структуры `IMAGE_IMPORT_DESCRIPTOR` (см. листинг 1.21). Команды `CALL`, вызывающие импортируемые функции, указывают на элементы этого массива непосредственно (так: `CALL DWORD PTR [adres]` или

так: MOV ESI,adres/CALL ESI) или вызывают в начале переходник (JMP DWORD PTR [adres]). Во время загрузки модуля загрузчик по именам или ординалам импортируемых функций определяет их истинные адреса в памяти и помещает эти адреса в данный массив. Массив же, на который указывает поле Characteristics, изменению при загрузке не подвергается. В *разд. 1.6.1* вы можете найти развернутый пример поиска имени импортируемой функции.

1.5.5. Таблица экспорта

Таблица экспорта необходима динамическим библиотекам для того, чтобы приложения могли корректно вызывать предоставляемые библиотекой функции. Как и в случае с таблицей импорта, следует воспользоваться массивом DataDirectory из структуры IMAGE_NT_HEADERS — секция .edata может в исполняемом модуле и отсутствовать. В данном случае нам понадобится самый первый элемент массива (значение индекса 0).

По указанному адресу располагается структура IMAGE_EXPORT_DIRECTORY. В ней расположена вся необходимая информация об экспортируемых функциях (листинг 1.23).

Листинг 1.23

```
struct IMAGE_EXPORT_DIRECTORY {
    DWORD   Characteristics;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    DWORD   Name;
    DWORD   Base;
    DWORD   NumberOfFunctions;
    DWORD   NumberOfNames;
    DWORD   AddressOfFunctions;
    DWORD   AddressOfNames;
    DWORD   AddressOfNameOrdinals;
}
```

Разберем поля структуры IMAGE_EXPORT_DIRECTORY:

- Characteristics — резерв; по-видимому, всегда 0;
- TimeDateStamp — время и дата создания экспортных данных или 0;

- `MajorVersion` — старшая часть версии таблицы экспорта; скорее всего, 0;
- `MinorVersion` — младшая часть версии таблицы экспорта; скорее всего, 0;
- `Name` — имя экспортирующего модуля; в принципе, может и не совпадать с именем файла;
- `Base` — начальный номер экспортируемой функции. Экспортируемые функции кроме имени имеют и номер, так что доступ к ним (импорт) можно осуществлять по этому номеру. Такой номер называют еще *ординалом* (ordinal);
- `NumberOfFunctions` — количество элементов в массиве адресов экспортируемых функций;
- `NumberOfNames` — количество элементов в массиве имен экспортируемых функций;
- `AddressOfFunctions` — относительный виртуальный адрес массива виртуальных адресов экспортируемых функций;
- `AddressOfNames` — относительный виртуальный адрес массива, в котором содержатся относительные виртуальные адреса имен экспортируемых функций;
- `AddressOfNameOrdinals` — относительный виртуальный адрес 16-битного массива (массив ординалов), содержащего значения индексов для массива адресов экспортируемых функций. Для получения ординала функции следует к значению индекса прибавить значение поля `Base`.

Для полного уяснения того, как можно получить информацию об экспортируемых функциях, следует понять, как соотносятся друг с другом три массива: массив адресов функций, массив адресов имен, массив ординалов. Последний массив является связующим звеном между двумя первыми массивами. Количество элементов в массиве имен равно количеству элементов в массиве ординалов. Поэтому для того чтобы получить адрес функции по ее имени, следует совершить следующие шаги:

1. Найти по имени функцию в массиве имен.
2. Взять индекс, по которому находится нужное имя в массиве имен, и найти в массиве ординалов элемент с таким значением индекса.
3. Взять значение найденного элемента в массиве ординалов, которое и будет служить индексом для массива адресов функций, и, обратившись к массиву адресов функций, получить нужный адрес.

Проанализируйте программу из *приложения* на предмет работы с таблицей экспорта и поэкспериментируйте с определением таблицы экспорта у различных программ и динамических библиотек.

1.5.6. Ресурсы

Как и в предыдущих случаях, для получения доступа к блоку ресурсов следует воспользоваться массивом DataDirectory из структуры IMAGE_NT_HEADERS. Нам понадобится элемент массива с индексом 2. В отличие от ранее рассмотренных объектов PE-модуля, секция ресурсов имеет древовидную структуру, в которой на практике используются четыре уровня. Кроме этого, все адреса, используемые внутри секции ресурсов, отсчитываются от начала секции ресурсов, т. е. не являются RVA. Это и понятно, ведь ресурсы загружаются в память по мере обращения к ним, а не во время загрузки самого модуля.

По сути, для того чтобы понять структуру ресурсов, нам понадобятся лишь две структуры, представленные в листингах 1.24 и 1.25.

Листинг 1.24

```
struct IMAGE_RESOURCE_DIRECTORY {
    DWORD   Characteristics;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    WORD    NumberOfNamedEntries;
    WORD    NumberOfIdEntries;
}
```

Разберем поля структуры IMAGE_RESOURCE_DIRECTORY:

- Characteristics — поле флагов, которое в настоящее время, по-видимому, не используется;
- TimeDateStamp — дата и время создания ресурсов;
- MajorVersion, MinorVersion — старшая и младшая части версии ресурсов; поля бесполезны;
- NumberOfNamedEntries — общее количество именованных ресурсов (имеющих имя);
- NumberOfIdEntries — общее количество ресурсов, заданных своим идентификатором.

Листинг 1.25

```
struct IMAGE_RESOURCE_DIRECTORY_ENTRY {
    ULONG   Name;
    ULONG   OffsetToData;
}
```

Поля структуры IMAGE_RESOURCE_DIRECTORY_ENTRY:

- Name — поле может интерпретироваться по-разному в зависимости от уровня и значение старшего бита; я оговорю ниже все эти случаи;
- OffsetToData — это поле является адресом, вычисленным относительно начала секции ресурсов; на том, что может указывать данный адрес, я остановлюсь отдельно.

Итак, перейдя по адресу, который указан во втором элементе (с индексом 2) массива DataDirectory, мы оказываемся в царстве ресурсов. Здесь начинается первый уровень. Лишний раз подчеркну, что если значение адреса равно нулю, то это может означать только, что блок ресурсов отсутствует.

Первый уровень

На самом верхнем (первом) уровне ресурсов располагается структура IMAGE_RESOURCE_DIRECTORY (см. листинг 1.24). Единственное поле, которое может дать нам возможность исследовать ресурсы, — это NumberOfIdEntries. На первом уровне это поле содержит количество типов ресурсов, хранящихся в заголовке PE. Поле же NumberOfNamedEntries на первом уровне вообще не имеет никакого значения. Итак, что же нам дает количество типов ресурсов? Оказывается, это ключ, потому что за структурой IMAGE_RESOURCE_DIRECTORY сразу следует массив структур IMAGE_RESOURCE_DIRECTORY_ENTRY (см. листинг 1.25), их количество как раз равно значению, которое хранится в поле NumberOfIdEntries, так что мы без труда прочтем их один за другим. Поле Name структуры IMAGE_RESOURCE_DIRECTORY_ENTRY на первом уровне содержит идентификатор типа ресурса. Идентификаторы типов ресурсов можно найти в файле winuser.h пакета Visual Studio .NET (листинг 1.26).

Листинг 1.26

```
#define RT_CURSOR         1
#define RT_BITMAP         2
#define RT_ICON           3
```

```
#define RT_MENU            4
#define RT_DIALOG          5
#define RT_STRING          6
#define RT_FONTDIR         7
#define RT_FONT            8
#define RT_ACCELERATOR     9
#define RT_RCDATA          10
#define RT_MESSAGETABLE    11
#define RT_GROUP_CURSOR    12
#define RT_GROUP_ICON      14
#define RT_VERSION         16
#define RT_DLGINCLUDE      17
#define RT_PLUGPLAY        19
#define RT_VXD             20
#define RT_ANICURSOR       21
#define RT_ANIICON         22
#define RT_HTML            23
#define RT_MANIFEST        24
```

Итак, на первом уровне мы узнали, сколько типов ресурсов имеется в модуле, и можем легко их идентифицировать.

Поле OffsetToData каждого из элементов массива указывает на структуры IMAGE_RESOURCE_DIRECTORY, но уже второго уровня.

Второй уровень

Второй уровень опять начинается со структур IMAGE_RESOURCE_DIRECTORY. Количество их равно количеству типов ресурсов в модуле *(см. конец предыдущего раздела)*. В них нас будут интересовать два поля: NumberOfNamedEntries и NumberOfIdEntries. Первое поле будет содержать количество именованных ресурсов, второе — количество ресурсов, заданных по идентификатору. Таким образом, сразу за каждой структурой IMAGE_RESOURCE_DIRECTORY на втором уровне будет следовать массив из структур IMAGE_RESOURCE_DIRECTORY_ENTRY, количество элементов в котором будет равно NumberOfNamedEntries + NumberOfIdEntries. На полях структуры IMAGE_RESOURCE_DIRECTORY_ENTRY, из которой состоит массив, следует остановиться особо. Поле Name теперь следует интерпретировать по-другому. Если старший бит этого поля равен 0, само поле представляет собой идентификатор ресурса. Если же старший бит равен 1, то остальные биты следует интерпретировать как смещение относительно начала блока ресурсов

имени данного ресурса. Причем структура имени такова: в начале идут два байта — длина имени в символах (именно в символах, а не в байтах), а далее — само имя, но в кодировке Unicode.

Обратимся опять к полю `OffsetToData`. В данном случае оно для каждой структуры `IMAGE_RESOURCE_DIRECTORY_ENTRY` второго уровня указывает на такую же структуру, но третьего уровня.

Третий уровень

Стало быть, разветвление закончилось вторым уровнем. Массив структур `IMAGE_RESOURCE_DIRECTORY_ENTRY` на третьем уровне соответствует аналогичным структурам второго уровня. Рассмотрим, как же теперь следует интерпретировать поля этой структуры на третьем уровне. Поле `Name` теперь определяет номер (идентификатор) языка ресурса. Все идентификаторы определены в заголовочном файле WINNT.H, они начинаются с префикса `LANG_`, и мы не будем их перечислять. Поле же `OffsetToData` опять указывает на структуру `IMAGE_RESOURCE_DIRECTORY_ENTRY`, но уже четвертого уровня.

Четвертый уровень

На четвертом уровне поле `Name` структуры `IMAGE_RESOURCE_DIRECTORY_ENTRY` определяет размер, который занимает двоичный образ ресурса. Адрес (относительно начала секции ресурсов, как обычно) области памяти, где располагается двоичное описание ресурса, определяется полем `OffsetToData`.

На этом я закончу описание ресурсов, заметив только, что программа из *приложения* анализирует лишь два уровня ресурсов. Впрочем, в большинстве случаев иного и не требуется.

1.5.7. Об отладочной информации

Описание структуры PE-модуля будет неполным, если мы, хотя бы кратко, не остановимся на отладочной информации. Программа из *приложения* сообщает только о наличии такой информации (отладочной информации и таблицы символов) и адресов (смещений), по которым она располагается.

Таблица символов

Положение *таблицы символов* определяется из файлового заголовка `FileHeader`. Поле `PointerToSymbolTable` как раз содержит относительный виртуальный адрес таблицы символов. Если поле равно нулю, то таблица

символов отсутствует. Что представляет собой таблица символов? Название, конечно, несколько неточное. Под символом здесь понимается идентификатор языка высокого уровня: переменная или функция. Таблица символов содержит: имя символа (имя переменной, функции), относительный виртуальный адрес символа, тип символа (переменной или функции), класс памяти символа (automatic, register, label и т. д.). Вся эта информация об идентификаторе упакована в структуру IMAGE_SYMBOL, которую можно найти в файле WINNT.H.

Отладочная информация

Говоря об отладочной информации, мы имеем в виду информацию о номерах строк программы. Эта информация хранится в PE-модуле в другой области, нежели таблица символов. Но чтобы выйти на эту информацию, придется немного потрудиться. Необходимо выйти на заголовок IMAGE_DEBUG_DIRECTORY. На него указывает шестой (с индексом 6) элемент массива DataDirectory из структуры IMAGE_NT_HEADERS. Если в структуре PE-файла содержится несколько типов отладочной информации, то для каждого типа имеется своя структура IMAGE_DEBUG_DIRECTORY. Поле TYPE этой структуры и определяет тип отладочной информации. Типы отладочной информации вы опять же можете найти в файле WINNT.H. Они заданы в константах IMAGE_DEBUG_TYPE_. Например, значение 1 соответствует отладочной информации формата COFF, значение 9 (IMAGE_DEBUG_TYPE_BORLAND) соответствует отладочной информации Borland, и т. д. Поле PointerToRawData структуры IMAGE_DEBUG_DIRECTORY, в случае если поле TYPE равно 1, должно содержать смещение от начала файла блока отладочной информации формата COFF. Именно там должна располагаться структура IMAGE_COFF_SYMBOLS_HEADER. Это ключевой момент. Структура содержит информацию и о таблице символов, на которую мы раньше вышли другим способом *(см. предыдущий раздел)*, и на таблицу номеров строк. Поле NumberOfSymbols должно содержать количество идентификаторов в символьной таблице. Это число будет в точности равно содержимому NumberOfSymbols поля в структуре IMAGE_FILE_HEADER (см. листинг 1.16). Поле LvaToFirstSymbol будет содержать смещение от начала структуры IMAGE_COFF_SYMBOLS_HEADER таблицы символов. Таким образом, мы вышли на таблицу символов другим, я бы сказал, более академичным способом. Наконец, поле LvaToFirstLinenumber содержит смещение от начала структуры COFF-таблицы номеров строк.

1.6. Об отладке и дизассемблировании программ, написанных на языке ассемблера

В данном разделе мы займемся языком ассемблера, поскольку отладка и дизассемблирование программ, написанных на этом языке, наиболее просты.

1.6.1. Примеры дизассемблирования

Разберем несколько примеров дизассемблирования, которые, на мой взгляд, помогут читателю быстро освоить этот процесс.

Пример поиска импортируемой функции

Рассмотрим чрезвычайно простой пример программы на языке ассемблера. Текст программы вы можете видеть в листинге 1.27.

Листинг 1.27

```
.586P
.MODEL FLAT,STDCALL
includelib f:\masm32\lib\user32.lib
EXTERN  MessageBoxA@16:NEAR
;сегмент данных
_DATA SEGMENT
TEXT1 DB 'No problem!',0
TEXT2 DB 'Message',0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
  PUSH OFFSET 0
  PUSH OFFSET TEXT2
  PUSH OFFSET TEXT1
  PUSH 0
  CALL MessageBoxA@16
  RETN
_TEXT ENDS
END START
```

5 Зак. 119

Программа из листинга 1.27 чрезвычайно проста. Ее единственная задача — вывести окно-сообщение `MessageBox`. Чтобы получить исполняемый модуль, выполним две команды:

```
ML  /c /coff  prog.asm
LINK /subsystem:console prog.obj
```

То, что мы создаем именно консольную программу, не имеет в данном случае никакого значения. Впрочем, читатель (из любопытства) может использовать для трансляции ключ `/subsystem:windows` и попытаться объяснить разницу в выполнении обеих программ.

В результате трансляции в каталоге появится исполняемый файл prog.exe. Собственно, для программиста, знакомого с ассемблером, это все достаточно очевидно. А вот дизассемблирование даже такой простой программы — куда более интересное занятие. Воспользуемся для дизассемблирования программой dumpbin.exe из пакета Visual Studio .NET. Выполним команду

```
dumpbin /disasm prog.exe > prog.txt
```

Содержимое текстового файла prog.txt представлено в листинге 1.28.

Листинг 1.28

```
Microsoft (R) COFF/PE Dumper Version 7.10.3077
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file r8.exe

File Type: EXECUTABLE IMAGE

  00401000: 6A 00              push       0
  00401002: 68 0C 30 40 00     push       40300Ch
  00401007: 68 00 30 40 00     push       403000h
  0040100C: 6A 00              push       0
  0040100E: E8 01 00 00 00     call       00401014
  00401013: C3                 ret
  00401014: FF 25 00 20 40 00  jmp        dword ptr ds:[00402000h]

  Summary

        1000 .data
        1000 .rdata
        1000 .text
```

Программа dumbin.exe оказалась вполне работоспособной и добросовестно выполнила дизассемблирование нашего модуля. Из текста мы легко узнаем вызов импортируемой функции `MessageBox`. Это в частности вытекает из значения параметров. Выполнив команду

```
dumpbin /rawdata /section:.data prog.exe > prog.txt
```

получим содержимое секции .data, где должны располагаться инициализированные данные (листинг 1.29).

Листинг 1.29.

```
RAW DATA #3
00403000: 4E 6F 20 70 72 6F 62 6C 65 6D 21 00 4D 65 73 73  No problem!.Mess
00403010: 61 67 65 00                                      age.
```

Сравнив адреса параметров из листинга 1.28 с данными из листинга 1.29, мы убеждаемся, что вызов `CALL` — это и есть вызов импортируемой функции `MessageBox`.

Однако мы разрешили не все вопросы, возникающие при просмотре листинга 1.28. Дело в том, что вызов осуществляется по адресу, где стоит команда `JMP`. Чтобы понять, что это значит, нам придется обратиться к *разд. 1.5.4*, где мы разбирали таблицу импорта. Напомню вкратце, о чем там шла речь. Таблица импорта состоит из массива структур `IMAGE_IMPORT_DESCRIPTOR` (см. листинг 1.21). Количество структур в массиве равно количеству используемых динамических библиотек. Речь, конечно, идет о неявном связывании. В этой структуре имеется поле `FirstThunk`, которое должно указывать на массив структур `IMAGE_THUNK_DATA32` (для каждой dll-библиотеки). Эти структуры, по сути, состоят из указателей на имена импортируемых функций. После загрузки исполняемого модуля вместо адресов имен функций загрузчик помещает сюда адреса самих функций в dll-библиотеке. Команда `jmp dword ptr ds:[00402000h]` вызывает импортируемую функцию, адрес которой должен находиться по адресу `00402000h`. Таким образом, можно сделать вывод, что виртуальный адрес `00402000h` — это виртуальный адрес элемента массива, на который указывает поле `FirstThunk`. Воспользовавшись программой из *приложения*, мы можем получить относительный виртуальный адрес и смещение для массива `IMAGE_THUNK_DATA32` (в программе он называется `AdresImpArray`). Относительный виртуальный адрес оказывается равным `2000h`. Здесь все верно, поскольку виртуальный адрес загрузки равен `400000h`. Смещение же оказывается равным `600h`, и мы можем в файле prog.exe найти массив структур `IMAGE_THUNK_DATA32`. Это можно сделать самой простой программой просмотра файлов в шестнадцатеричном представлении из файлового менеджера far.exe. Оказывается, что по адресу `600h` располагаются байты `38 20 00 00`, т. е. число `2038h`. Это число есть не что

иное, как относительный виртуальный адрес (за минусом двух) имени импортируемой функции MessageBox, т. е. истинным относительным виртуальным адресом имени функции, а затем после загрузки самой функции является 203Ah. Опять воспользовавшись программой из *приложения*, мы убеждаемся, что все верно, и смещение имени функции в файле prog.exe должно располагаться по адресу 63Ah. Обратившись к файлу prog.exe, убеждаемся, что действительно по данному смещению располагается строка MessageBoxA.

Быть может, наши рассуждения с использованием нескольких программ показались читателю сложными. Давайте попытаемся проделать те же действия, используя только программу hiew.exe. Данная программа является одним из лучших hex-редакторов, незаменимым инструментом при исправлении исполняемых модулей. Она также обладает возможностями дизассемблирования PE-модулей. Я буду использовать версию программы 6.11. Итак, загрузим в программу hiew.exe наш исполняемый модуль prog.exe. Вот что мы обнаружим по адресу 401000h в режиме дизассемблирования (листинг 1.30).

Листинг 1.30

```
.00401000:  6A00          push 00
.00401002:  680C304000    push 000403000C
.00401007:  680C304000    push 0004030000
.0040100C:  6A00          push 00
.0040100E:  E801000000    call .000401014
.00401013:  C3            retn
.00401014:  FF2500204000  jmp MessageBoxA
```

Как видим, программа hiew.exe оказалась более продвинутой, чем dumpbin.exe, поскольку она распознала вызов MessageBoxA. Из кода команды jmp определяем адрес перехода. Это будет 402000h, как и следовало ожидать (не забывайте, как хранятся байты целых чисел, и что первые два байта кода команды — сам код и байт MOD R/M, *см. разд. 1.4*). Перейдем теперь в шестнадцатеричный режим просмотра и обратимся к полученному адресу. Там, как и следовало ожидать, мы обнаруживаем байты 38 20 00 00, т. е. число 2038h. Это относительный виртуальный адрес. Чтобы получить виртуальный адрес, добавим к нему адрес загрузки модуля, который равен 400000h. Адрес же строки, которая должна содержать имя импортируемой функции, т. е. MessageBoxA, получается так: 400000h + 2038h + 2h = 40203Ah. Обратимся к полученному адресу и, разумеется, обнаружим там искомое имя.

Интересно посмотреть, что же получится, если программа будет откомпилирована при помощи TASM32. Для этого в программе имя MessageBoxA@16

следует заменить на имя `MessageBoxA`, а библиотеку импорта user32.lib на библиотеку import32.lib от Borland. Для компилирования используем следующие команды:

```
tasm32 /ml  prog.asm
tlink32 -ap  prog.obj
```

После компиляции запустим программу hiew.exe и загрузим в нее исполняемый модуль prog.exe (обратите внимание, что компилятор от Borland создает менее компактные исполняемые модули, чем аналогичный от Microsoft). В листинге 1.31 — дизассемблированный текст. Сравните его с тем, что представлен в листинге 1.30. Как видим, практически идентичный текст, правда, адресация немного другая.

Листинг 1.31

```
.00401000:  6A00          push 00
.00401002:  680C204000    push 00040200C
.00401007:  680C204000    push 0004020000
.0040100C:  6A00          push 00
.0040100E:  E801000000    call . 000401014
.00401013:  C3            retn
.00401014:  FF2530304000  jmp MessageBoxA
```

Обратимся по адресу `403030h`, это адрес элемента массива, указывающего на имя импортируемой функции. Там располагается цепочка байтов `44 30 00 00`, т. е. по адресу `40000h` + `3044h` должно присутствовать имя импортируемой функции. Так и есть, имя находится именно там.

Некоторые сложности в распознавании исполняемого кода

Хотя, казалось бы, при дизассемблировании исполняемых модулей, написанных на языке ассемблера, не должно быть особых проблем, все же коекакие сложности возникают. И мы о них будем говорить.

Рассмотрим программу из листинга 1.32. Вначале откомпилируем ее с помощью ассемблера MASM32.

Листинг 1.32

```
.586P
.MODEL FLAT,STDCALL
includelib f:\masm32\lib\user32.lib
```

```
EXTERN  MessageBoxA@16:NEAR
;сегмент данных
_DATA SEGMENT
TEXT1 DB 'No problem!',0
TEXT2 DB 'Message',0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
  PUSH OFFSET 0
  PUSH OFFSET TEXT2
  PUSH OFFSET TEXT1
  PUSH 0
  CALL MessageBoxA@16
  RETN
       DB 50
l1:
       RETN
_TEXT ENDS
END START
```

Программа из листинга 1.32, конечно, чрезвычайно странная. К чему метка l1, если на нее нет никаких переходов? Погодите, метка нам еще пригодится. А к чему последовательность DB 50/RETN, она же не выполняется? Терпение, обо всем по порядку. Мне просто интересно, как на такой фрагмент среагируют наши дизассемблеры. Как я и предполагал (надеюсь, что и вы тоже), весь следующий за первой командой RETN фрагмент кода дизассемблерами будет понят неправильно. Да и как, скажите, им его понять, если это просто последовательность байтов 32 C3, которая соответствует команде XOR AL,BL. Так оно и есть, все дизассемблеры, в том числе и легендарный IDA Pro, посчитали, что за командой RETN стоит команда XOR AL,BL.

Замечание

Удивил меня отладчик (и дизассемблер тоже) OllyDbg. После загрузки он показал последовательность DB 50/RETN. "Мистика", — подумал я и, на секунду уверовав в величие отладчика, заменил последовательность байтов на одну команду XOR AL,BL. Но он тупо твердил, что это DB 50/RETN, и я был разочарован.

Теперь видоизменим нашу программу всего одной командой MOV EBX,OFFSET l1 (листинг 1.33). Разумеется, она не имеет никакого смысла, но это только для нас. А что скажут наши дизассемблеры?

Листинг 1.33

```
.586P
.MODEL FLAT,STDCALL
includelib f:\masm32\lib\user32.lib
EXTERN  MessageBoxA@16:NEAR
;сегмент данных
_DATA SEGMENT
TEXT1 DB 'No problem!',0
TEXT2 DB 'Message',0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
  MOV  EBX,OFFSET l1
  PUSH OFFSET START
  PUSH OFFSET 0
  PUSH OFFSET TEXT2
  PUSH OFFSET TEXT1
  PUSH 0
  CALL MessageBoxA@16
  POP  EDX
  ADD  EDX,l1-START
  CALL EDX
  RETN
  DB 50
l1:
   RETN
_TEXT ENDS
END START
```

Проверим откомпилированный код с помощью трех дизассемблеров. Программа hiew.exe ничего не замечает, т. е. отношение ее к коду после команды RETN не изменилось. Аналогично ведет себя весьма уважаемый (не только мной) дизассемблер W32Dasm. А вот IDA Pro (ну, это же IDA Pro) сразу реагирует на нашу команду. Для большей наглядности приведу листинг-фрагмент из IDA Pro (листинг 1.34).

Листинг 1.34

```
.text:00401000 ; ------- S U B R O U T I N E ---------------------------
.text:00401000
.text:00401000
.text:00401000                public start
.text:00401000 start          proc near          ; DATA XREF: start+5?o
.text:00401000                mov    ebx, offset nullsub_1
.text:00401005                push   offset start
.text:0040100A                push   0              ; uType
.text:0040100C                push   offset Caption ; lpCaption
.text:00401011                push   offset Text    ; lpText
.text:00401016                push   0              ; hWnd
.text:00401018                call   MessageBoxA
.text:0040101D                pop    edx
.text:0040101E                add    edx, 28h
.text:00401024                call   edx
.text:00401026                retn
.text:00401026 start          endp
.text:00401026
.text:00401026 ;--------------------------------------------------------
.text:00401027                db 32h
.text:00401028 ; [00000001 BYTES: COLLAPSED FUNCTION nullsub_1. PRESS
KEYPAD "+" TO EXPAND]
```

Обращаю ваше внимание, дорогие друзья, как дизассемблирована бывшая команда `MOV EBX,OFFSET l1`. Имя `nullsub_1` означает, что данная метка указывает на процедуру, состоящую всего из одной команды `RETN` — пустой процедуры (от англ. *null* — пустой). Комментарий по адресу `00401028` означает, что процедура находится в *свернутом состоянии* (collapsed). Для разворачивания процедуры, т. е. для просмотра ее текста достаточно нажать клавишу <+> на дополнительной клавиатуре. В данном случае разворачиваемая процедура состоит всего из одной команды `RETN`.

Итак, дизассемблер IDA Pro "отделил зерна от плевел", т. е. отделил команду `RETN` от кода `32H`. Хорошо это или плохо? Вы удивлены, что я задаю такой вопрос? А представьте себе, что в исходном тексте просто была команда `MOV EBX,N`, где `N` — некое число. И вот оказалось, что данное число попадает в адресный промежуток, но при этом вовсе не является адресом какой-либо команды. Но на этом основании дизассемблер делает вывод, что по указанному адресу располагается некая процедура. Конечно, такого рода ошибка

не несет в себе ничего серьезного, ведь никакого перехода на данный адрес нет. Впрочем, переходов на процедуры окон также нет, но там адрес определяется по вызову одной из функций API *(см. разд. 1.3)*. Но, так или иначе, такое ошибочное обнаружение процедуры не несет в себе ничего серьезного. А вот если все же это действительно оказался адрес некоторый команды, куда затем "тайно" (о "тайных" переходах поговорим далее) будет произведен переход, то это может вам здорово помочь при анализе кода. Авторы IDA Pro довольно логично посчитали, что число, попавшее в диапазон адресов команд, с большой вероятностью является именно адресом, и сделали, на мой взгляд, правильный выбор.

Продолжим наши экспериментальные исследования. Заменим команду `MOV EBX,OFFSET l1` на `CALL l1`. Как на это посмотрят наши дизассемблеры? Разумеется, IDA Pro отслеживает адрес процедуры и отмечает его в листинге. Программа hiew.exe, напротив, по-прежнему не распознает процедуры, хотя и показывает команду `CALL`. Впрочем, что же с нее взять, основное предназначение этой программы — совсем не дизассемблирование, это, так сказать, побочный продукт. Что касается W32Dasm, то на этот раз дизассемблер не ударил лицом в грязь. Вот фрагмент листинга, который генерирует эта программа (листинг 1.35).

Листинг 1.35

```
//******************** Program Entry Point ********
:00401000 E823000000              call 00401028
:00401005 6800104000              push 00401000
:0040100A 6A00                    push 00000000

* Possible StringData Ref from Data Obj ->"Message"
                                  |
:0040100C 680C304000              push 0040300C

* Possible StringData Ref from Data Obj ->"No problem!"
                                  |
:00401011 6800304000              push 00403000
:00401016 6A00                    push 00000000

* Reference To: user32.MessageBoxA, Ord:019Dh
                                  |
:00401018 E80D000000              Call 0040102A
:0040101D 5A                      pop edx
```

```
:0040101E 81C228000000          add edx, 00000028
:00401024 FFD2                  call edx
:00401026 C3                    ret


:00401027 32                    BYTE 32h


* Referenced by a CALL at Address:
|:00401000
|
:00401028 C3                    ret
```

Как видим из листинга, дизассемблер W32Dasm распознает адрес 00401028h, как адрес процедуры ("Referenced by a CALL at Address 00401000" — "Ссылка по команде CALL с адреса 00401000").

Итак, дорогой читатель, как видно из приведенных примеров, в дизассемблировании кода есть определенные трудности. Никакой дизассемблер не сможет исчерпывающе проанализировать код, и исследователям кода хватит работы.

"Тайные переходы" и тайны переходов

Что я называю "тайным переходом"? Существуют следующие, наиболее часто используемые, команды передачи управления: JMP, группа условных переходов JXX, CALL, RETN, LOOP. Так вот, с помощью одной команды перехода можно имитировать совсем другую команду из этой же группы. Единственной целью, с которой это может делаться, — запутать тех, кто анализирует код. Вот сейчас и займемся этим вопросом, чтобы уметь противостоять данным уловкам.

Обратимся к команде JMP. Это самая простая команда из записанного выше списка, конечно, если рассматривать переходы в рамках плоской модели памяти, с которой мы работаем в данной книге. Казалось бы, все предельно просто. Происходит переход по указанному адресу. При этом содержимое всех регистров (кроме EIP) не меняется. Но ведь кроме стандартного перехода JMP l1, где l1 — просто метка, есть еще косвенные переходы:

- JMP DWORD PTR [*lo*], где переменная *lo* содержит некий адрес перехода;
- JMP EBX, где регистр EBX содержит адрес перехода;
- JMP DWORD PTR [EBX], где регистр EBX содержит адрес переменной, которая в свою очередь содержит адрес перехода.

Как, например, быть, если вы видите команду JMP EAX, а что содержится в регистре EAX — не знаете? Это содержимое могло быть сформировано за сотни команд от данной команды. В этой ситуации никакой дизассемблер вам не поможет. Есть лишь два выхода: вручную анализировать текст дизассемблированной программы или прибегнуть к отладчику. Узнав, наконец, какой же адрес находится в регистре, вы можете снова обратиться к дизассемблеру и написать в комментарии это значение. Такая функция уже давно реализована во многих современных дизассемблерах. Впрочем, мы забегаем вперед. В *главе 2*, когда мы будем рассматривать современные дизассемблеры, поговорим и о таких возможностях. Для нас сейчас важно уяснить суть проблемы и подходы к ее решению.

Проблема, однако, усложняется тем, что любая из перечисленных выше команд может "прикинуться" совсем другой. Например, команда LOOP вполне может играть роль короткого перехода (127 байтов вперед и 128 байтов назад), а не являться признаком наличия цикла.

Вот некоторые примеры (листинг 1.36).

Листинг 1.36

```
.586P
.MODEL FLAT,STDCALL
includelib f:\masm32\lib\user32.lib
EXTERN  MessageBoxA@16:NEAR
_DATA SEGMENT
;здесь хранится просто адрес
mem1  DD OFFSET l2
TEXT1 DB 'No problem!',0
TEXT2 DB 'Message',0
_DATA ENDS
_TEXT SEGMENT
START:
  MOV  EAX,mem1
;две следующие команды эквивалентны просто JMP l2
  PUSH EAX
  RETN
l1:
  RETN
l2:
  PUSH OFFSET 0
  PUSH OFFSET TEXT2
```

```
  PUSH OFFSET TEXT1
  PUSH 0
  CALL MessageBoxA@16
  RETN
_TEXT ENDS
END START
```

Программа из листинга 1.36 демонстрирует необычное для команды RET применение. Сочетание команд PUSH/RET оказывается эквивалентно команде JMP. Можно поступить и еще изощреннее, например, так:

```
MOV  EAX,mem1
SUB  ESP,4
MOV  DWORD PTR [ESP],EAX
RETN
```

И в результате опять обычный переход на адрес l2. При этом команды могут быть перемешаны с другими командами, и попробуй тогда разберись, куда произошел переход. Самое главное здесь то, что трюки с переходами по адресам, которые хранятся в стеке, можно усложнять до бесконечности, ведь в стек можно поместить произвольное количество адресов переходов, а команды можно перемешивать в любом порядке. Так уж устроен ассемблер, что возможности в этом плане в нем просто безграничны.

Аналогична ситуация с условными переходами. Дело в том, что часто проверить, выполняется ли данное условие или нет путем просто анализа текста, практически невозможно. В результате совершенно неясно, по какой ветке пойдет выполнение программы, и будет ли вообще выполняться одна из веток. Последовательности команд типа

```
...
CMP EAX,100
JA l1
...
l1:
```

часто очень сложно разгадать, т. к. отследить, что может находиться в регистре EAX, иногда весьма трудно. По сути, команда JA может играть просто роль команды JMP, поскольку на деле число в регистре EAX всегда может оказываться больше 100, а часть программы, которая идет после команды JA, не иметь никакого смысла. Здесь только отладчик и может помочь. Хотя 100-процентной уверенности все равно не будет, т. к. всегда можно считать, что возможна отличная от нуля вероятность выполнения программы по другому пути.

Прием, о котором я сейчас вам расскажу, называют *перекрытием кода.* Кратко суть приема в следующем. Часть кода команды может стать самостоятельной командой, смысл которой разгадать не всегда просто. Вот фрагмент программы:

```
MOV  AX,015EBH
JMP  $-2
PUSH OFFSET 0
PUSH OFFSET TEXT2
PUSH OFFSET TEXT1
PUSH 0
CALL MessageBoxA@16
l1:
RETN
```

Не так-то просто сообразить, что код `015EBH` — это просто `JMP SHORT l1`, и что в действительности команда `MOV AX,015EBH` всего лишь маскирует этот переход на метку `l1`.

Использование отладочной информации

Мы сейчас обсуждали возможность запутывания следов, другими словами, защиту от тех, кто будет пытаться анализировать код с разными намерениями. Но есть ведь и другая сторона медали. Довольно часто требуется дизассемблировать собственную программу, чтобы разобраться, как она работает, понять причину тех или иных ошибок. Для этого часто используется отладочная информация *(см. разд. 1.5.7)*.

Многие современные отладчики и дизассемблеры хорошо разбираются в отладочной информации и правильно реконструируют программу. Что касается языка ассемблера, то, к сожалению, использование отладочной информации, на мой взгляд, — не слишком эффективно и в основном касается имен переменных. В принципе переменные и так не плохо идентифицируются таким дизассемблером, как IDA Pro, который только разве не может узнать истинное имя переменной, если модуль не содержит отладочной информации.

Для того чтобы включить отладочную информацию при трансляции при помощи MASM32, следует в командной строке ml.exe указать ключ `/Zi` — включить полную отладочную информацию, а в командной строке link.exe — ключ `/DEBUG`. При этом отладочная информация добавляется в PDB-файл (файл имеет имя транслируемого модуля и расширение pdb, program data base). Можно указать ключ `/PDB:NONE`, тогда вся отладочная информация будет помещена в исполняемый модуль. Наконец, можно указать тип отладочной информации: `/DEBUGTYPE:{CV|COFF}` — `CV` (тип для отлад-

чика CodeView) или тип COFF. Аналогично при использовании ассемблера TASM можно поместить отладочную информацию в исполняемый модуль. Для этого в командной строке tasm32.exe указываем ключ /zi (вся отладочная информация), а в командной строке tlink32.exe — ключ /v. При выполнении данных действий вся информация о переменных, а также действиях с ними будет помещена в исполняемый модуль и, следовательно, может быть использована дизассемблерами и отладчиками. При этом замечу, что информация сохраняется даже о переменных, которые никак не используются в программе.

1.6.2. О динамическом изменении исполняемого кода

Конечно, модифицируемый код с одной стороны противоречит канонам программирования, по которым код — это код, и его следует исполнять, а данные — это данные, и их следует читать, а также при желании модифицировать. Но ведь с другой стороны есть принцип фон Неймана, при грубой трактовке которого нет принципиальной разницы между данными и кодом — все это лишь последовательность байтов или битов (как кому нравится). А потом, модификация кода — это великолепный прием, позволяющий сокрыть истинные намерения программы.

Тот, кто программировал в операционной системе MS-DOS, знает, что модификация кода во время исполнения — дело совсем простое. Там вы можете менять содержимое ячейки вне зависимости от содержания в ней данных или кода. В операционной системе Windows код напрямую модифицировать запрещено. Нельзя исполнять и код, расположенный в сегменте данных или динамической области памяти. Для того чтобы это делать, программа должна исполняться в нулевом кольце защиты. Для обычной программы вроде бы все пути к модификации собственного кода закрыты. Однако выход есть, и не один. И это мы обсудим в данном разделе.

Исполнение кода в стеке

Исполнение кода в стеке, наверное, — самый оптимальный способ самомодификации программы. Дело в том, что страницы памяти, которые отводятся под стек, имеют атрибуты, позволяющие не только читать и писать туда данные, но и исполнять там код. Разумеется, при переносе код можно модифицировать. Наконец, ассемблерные команды можно хранить и в сегменте данных, а затем перенести в стек и там выполнить. Использовать стек в языке высокого уровня хоть и можно, но с целым рядом оговорок. А на ассемблере это можно сделать без особых трудностей. Впрочем, кое-какие проблемы могут появиться и здесь.

Итак, имеем следующую консольную программу (листинг 1.37).

Листинг 1.37

```
.586P
.MODEL FLAT,STDCALL
includelib f:\masm32\lib\user32.lib
EXTERN MessageBoxA@16:NEAR
;--------------------------------------------------
_DATA SEGMENT
TEXT1 DB 'Я в стеке!',0
TEXT2 DB 'Сообщение из стека',0
_DATA ENDS
_TEXT SEGMENT
START:
;вызвать процедуру
CALL PROC1
RETN
PROC1 PROC
PUSH 0
PUSH OFFSET TEXT2
PUSH OFFSET TEXT1
PUSH 0
CALL MessageBoxA@16
RETN
PROC1 ENDP
_TEXT ENDS
END START
```

Пусть программа называется prog.asm. Откомпилируем ее:

```
ML /c /coff prog1
LINK /SUBSYSTEM:CONSOLE prog1.obj
```

В результате появится исполняемый модуль prog.exe, при выполнении которого на экране появится соответствующее сообщение.

Попробуем теперь решить проблему "в лоб". Скопируем содержимое процедуры `PROC1` в стек и запустим ее там. Вот эта программа (листинг 1.38).

Листинг 1.38

```
.586P
.MODEL FLAT,STDCALL
includelib f:\masm32\lib\user32.lib
EXTERN MessageBoxA@16:NEAR
;----------------------------------------------------
_DATA SEGMENT
TEXT1 DB 'Я в стеке!',0
TEXT2 DB 'Сообщение из стека',0
_DATA ENDS
_TEXT SEGMENT
START:
;подготовить стек
MOV EBP,ESP
MOV ECX,OFFSET L1
SUB ECX,PROC1
;выделить место в стеке
SUB ESP,ECX
;скопировать код
MOV EDI,ESP
LEA ESI,PROC1
CLD
REP MOVSB
;вызвать процедуру из стека
CALL ESP
;восстановить стек
MOV ESP,EBP
RETN
PROC1 PROC
PUSH 0
PUSH OFFSET TEXT2
PUSH OFFSET TEXT1
PUSH 0
CALL MessageBoxA@16
RETN
PROC1 ENDP
_TEXT ENDS
END START
```

Однако нас ждет разочарование. После запуска программы появляется сообщение ОС об ошибке. Попробуем разобраться, в чем здесь дело, обратившись к отладчику OLLYDBG. Запустив программу под управлением отладчика, выполним ее в пошаговом режиме. Дойдя до команды `CALL ESP`, нажмем клавишу <F7> и окажемся в том месте стека, куда была скопирована процедура. На первый взгляд, код скопировался корректно (см. фрагмент ниже). Однако что это?

```
000CFFB0  6A  00        PUSH  0
000CFFB2  68  0B304000  PUSH  40300B
000CFFB7  68  00304000  PUSH  403000
000CFFBC  6A  00        PUSH  0
000CFFBE  E8  02000000  CALL  000CFFC5
000CFFC3  C3            RETN
```

Адрес, по которому осуществляется вызов процедуры, находится здесь же в стеке. Откуда здесь взяться какому-либо переходу на `MessageBox`? Все очень просто. В команде `CALL MessageBoxA@16` транслятор ассемблера подставляет относительные адреса. Вот оно что! Что же делать? Неужели придется корректировать адрес при переносе в стек? К счастью, процедуру можно вызвать и так: `LEA EBX,MessageBoxA@16`, `CALL EBX`. Попробуем проверить. Перепишем программу (листинг 1.39).

Листинг 1.39

```
.586P
.MODEL FLAT,STDCALL
includelib f:\masm32\lib\user32.lib
EXTERN MessageBoxA@16:NEAR
;-----------------------------------------------
_DATA SEGMENT
TEXT1 DB 'Я в стеке!',0
TEXT2 DB 'Сообщение из стека',0
_DATA ENDS
_TEXT SEGMENT
START:
;подготовить стек
MOV EBP,ESP
MOV ECX,OFFSET L1
SUB ECX,PROC1
;выделяем место в стеке
SUB ESP,ECX
```

```
;скопировать код
MOV EDI,ESP
LEA ESI,PROC1
CLD
REP MOVSB
;вызвать процедуру из стека
CALL ESP
;восстановить стек
MOV ESP,EBP
RETN
PROC1 PROC
PUSH 0
PUSH OFFSET TEXT2
PUSH OFFSET TEXT1
PUSH 0
LEA EBX,MessageBoxA@16
CALL EBX
RETN
PROC1 ENDP
L1:
_TEXT ENDS
END START
```

После трансляции запустим программу. На этот раз ошибка не появилась, но окно MessageBox вышло какое-то неожиданное. Точнее, без всяких надписей. Попытка выполнить программу под управлением отладчика ни к чему не приводит. Вернее, мы убеждаемся, что на этот раз вызов MessageBox будет осуществляться по правильному адресу. В чем же здесь дело? Проведем следующий эксперимент. Заменим в программе команду CALL ESP на CALL PROC1, т. е. проверим, а будет ли выполняться сама процедура? И о... удивление! Результат получается аналогичный. Что же вызвало ошибку? Поскольку раньше процедура выполнялась нормально, попробуем убирать по одной команде, которые мы добавили для копирования процедуры в стек, и выясним, как команда приводит, в конечном итоге, к ошибке. Оказывается, такой командой является SUB ESP,ECX. Ну, тут уж подозрение начинает закрадываться к нам в душу. Что же в этой команде плохого? Сплошь и рядом такие команды используют и ассемблерщики, и компиляторы. Значение, которое хранится в ECX, не велико, чтобы выйти за границы стека, да и ошибка в этом случае была бы другой. Наконец, доходит: адрес в стеке должен быть кратен 4. У нас, очевидно, это условие не выполняется. Попробуем откорректировать содержимое ECX прежде, чем вычитать его из ESP. Это

можно сделать разными способами. Например, так: SHL ECX,2, т. е. умножит содержимое на четыре. А можно так (если четыре для вас — слишком большое число): AND ECX,FFFFFFFCH/SHL ECX,1. В обоих случаях результат будет положительным, т. е. код в стеке заработает верно. Но проще использовать директиву ALIGN 4, так чтобы адреса PROC1 и L1 оказались выровнены по двойному слову. Вот окончательный вариант программы (листинг 1.40).

Листинг 1.40

```
.586P
.MODEL FLAT,STDCALL
includelib f:\masm32\lib\user32.lib
EXTERN MessageBoxA@16:NEAR
;---------------------------------------------------
_DATA SEGMENT
TEXT1 DB 'Я в стеке!',0
TEXT2 DB 'Сообщение из стека',0
_DATA ENDS
_TEXT SEGMENT
START:
;подготовить стек
MOV EBP,ESP
MOV ECX,OFFSET L1
SUB ECX,PROC1
;выделяем место в стеке
SUB ESP,ECX
;скопировать код
MOV EDI,ESP
LEA ESI,PROC1
CLD
REP MOVSB
;вызвать процедуру из стека
CALL ESP
;восстановить стек
MOV ESP,EBP
RETN
ALIGN 4
PROC1 PROC
```

```
PUSH 0
PUSH OFFSET TEXT2
PUSH OFFSET TEXT1
PUSH 0
LEA EBX,MessageBoxA@16
CALL EBX
RETN
PROC1 ENDP
ALIGN 4
L1:
_TEXT ENDS
END START
```

Итак, все достаточно просто, если следовать правилу: вызов процедур с помощью регистра и выравнивание кода по границе, кратной четырем.

Существует и еще одна проблема. Как быть с переходами? Если переход использует четырехбайтовый адрес, который должен находиться в перемещаемом фрагменте, то код в стеке не будет работать корректно. Но и здесь имеется очень простое решение: все такие переходы должны быть короткими (SHORT). Причем ничего, собственно, делать и не надо, поскольку ассемблер автоматически делает все переходы короткими, если они производятся в пределах 128 байтов. Вам надо только обеспечить, чтобы все нужные переходы и вызовы процедур осуществлялись в этом промежутке.

Используем функцию *WriteProcessMemory*

Еще один способ модификации кода во время исполнения — это использование API-функции `WriteProcessMemory`. С ее помощью можно писать данные в адресное пространство процесса. Область, куда предполагается писать, должна быть доступна для записи, в противном случае записи не произойдет, и функция возвратит нулевое значение (в случае успешной записи функция возвращает 0). Рассмотрим подробно параметры данной функции:

- 1-й параметр — дескриптор процесса, в память которого мы намереваемся писать;
- 2-й параметр — адрес в памяти процесса, куда мы намереваемся писать;
- 3-й параметр — указатель на буфер с данными, откуда будут браться данные для записи в память процесса;
- 4-й параметр — количество байтов, которые будут записаны;

- 5-й параметр — указатель на переменную, куда будет помещено количество байтов, записанных в память процесса. Если параметр равен 0, то он будет проигнорирован.

Как мы уже сказали, прежде чем писать в память процесса, мы должны получить дескриптор процесса. Для этого достаточно открыть процесс при помощи функции `OpenProcess`. Данная функция используется всякий раз, когда какая-то другая функция требует для выполнения дескриптор процесса. Разберем параметры функции:

- 1-й параметр — желаемый уровень доступа к процессу. Все уровни доступа выражены константами и перечислены в документации и заголовочных файлах. Имена этих констант начинаются с префикса `PROCESS_`. Нам понадобится комбинация двух констант `PROCESS_VM_OPERATION` и `PROCESS_VM_WRITE`;
- 2-й параметр — принимает два значения: 1, и тогда дескриптор может наследоваться, и 0, и тогда дескриптор не может наследоваться;
- 3-й параметр — идентификатор процесса, который мы хотим открыть.

Наконец, последнее, что следует отметить, — это как мы получим идентификатор процесса. Поскольку мы рассматриваем конкретную задачу записи в собственный код, то можно использовать API-функцию `GetCurrentProcessId`. Это функция без параметров и возвращает идентификатор вызывающего его процесса.

Итак, все объяснения сделаны, а пример программы, которая модифицирует собственный код, представлен в листинге 1.41. В консольной программе по адресу `RETE` записывается код `C3H`. Если это не сделать, то будет выполняться бесконечный цикл, и программа никогда не закончит свою работу (без воздействий извне).

Листинг 1.41

```
.586P
.MODEL FLAT,STDCALL
PROCESS_VM_OPERATION  =  0008H
PROCESS_VM_WRITE      =  0020H
PROCESS_VM_OW         =  PROCESS_VM_OPERATION OR PROCESS_VM_WRITE

includelib f:\masm32\lib\user32.lib
includelib f:\masm32\lib\kernel32.lib
EXTERN OpenProcess@12:NEAR
```

```
EXTERN WriteProcessMemory@20:NEAR
EXTERN GetCurrentProcessId@0:NEAR
;-------------------------------------------------
_DATA SEGMENT
OPC  DB 0C3H
_DATA ENDS
_TEXT SEGMENT
START:
  CALL GetCurrentProcessId@0
;в EAX идентификатор текущего процесса
  PUSH EAX
  PUSH 1
  PUSH PROCESS_VM_OW
  CALL OpenProcess@12
;в EAX дескриптор открытого процесса
  PUSH 0
  PUSH 1
  PUSH OFFSET OPC
  PUSH OFFSET RETE
  PUSH EAX
  CALL WriteProcessMemory@20
RETE:
  JMP  RETE
  RETN
_TEXT ENDS
END START
```

Замечание

После того как дескриптор какого-либо объекта был использован, его следует закрыть при помощи функции `CloseHandle`**, но в нашем случае при выходе из программы система все равно закрывает все дескрипторы.**

Конечно, использование функции `WriteProcessMemory` имеет ряд недостатков по отношению к исполнению кода в стеке. Прежде всего, с помощью данной функции вы исправляете код текущего процесса, но не можете увеличить объем памяти, чтобы добавить новый код. Кроме этого, исполнение кода в стеке обладает, я бы сказал, большей скрытностью, нежели использование функции `WriteProcessMemory`, которую легко обнаружит любой мало-мальски разбирающийся исследователь кода.

Используем функцию *VirtualProtectEx*

Вместо того чтобы писать в память процесса с помощью функции `WriteProcessMemory`, можно воспользоваться API-функцией `VirtualProtectEx` и разрешить доступ к нужным байтам (страницам, на которых располагаются байты), а потом воспользоваться обычной командой `MOV`.

В листинге 1.42 представлена программа, по строению такая же, как из листинга 1.41, но в данном случае используется функция `VirtualProtectEx`. Как и в предыдущем случае, по адресу `RETE` записывается байт `C3H`, но теперь для этого применяется простая команда `MOV`.

Листинг 1.42

```
.586P
.MODEL FLAT,STDCALL
PROCESS_VM_OPERATION  = 0008H
PROCESS_VM_WRITE      = 0020H
PROCESS_VM_OW         = PROCESS_VM_OPERATION OR PROCESS_VM_WRITE
PAGE_WRITECOPY        = 8
PAGE_EXECUTE          = 10h
includelib f:\masm32\lib\user32.lib
includelib f:\masm32\lib\kernel32.lib
;импортируемые функции
EXTERN OpenProcess@12:NEAR
EXTERN FlushInstructionCache@12:NEAR
EXTERN VirtualProtectEx@20:NEAR
EXTERN GetCurrentProcessId@0:NEAR
;-----------------------------------------------
_DATA SEGMENT
HANDLE  DD ?
NN      DD ?
_DATA ENDS
_TEXT SEGMENT
START:
  CALL GetCurrentProcessId@0
;открыть текущий процесс
  PUSH EAX
  PUSH 1
  PUSH PROCESS_VM_OW
  CALL OpenProcess@12
```

```
;разрешить копирование байта по адресу RETE
  MOV  HANDLE,EAX
  PUSH OFFSET NN
  PUSH PAGE_WRITECOPY
  PUSH 1
  PUSH OFFSET RETE
  PUSH EAX
  CALL VirtualProtectEx@20
;изменяем байт по адресу RETE
  LEA  EAX,RETE
  MOV  BYTE PTR [EAX],0C3H
;возвращаем байту первоначальный атрибут
  PUSH OFFSET NN
  PUSH PAGE_EXECUTE
  PUSH 1
  PUSH OFFSET RETE
  PUSH HANDLE
  CALL VirtualProtectEx@20
;сбрасываем кэш
  PUSH 1
  PUSH OFFSET RETE
  PUSH HANDLE
  CALL FlushInstructionCache@12
RETE:
  JMP  RETE
  RETN
_TEXT ENDS
END START
```

Разберем параметры функции `VirtualProtectEx`:

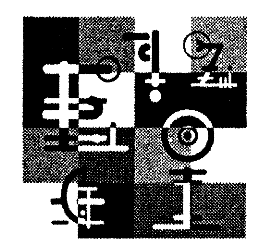
- 1-й параметр — дескриптор процесса, память которого мы намереваемся модифицировать;
- 2-й параметр — адрес области памяти, атрибут которой будем изменять;
- 3-й параметр — размер изменяемой области. При этом изменяется атрибут у всех страниц памяти, содержащие байты изменяемой области;
- 4-й параметр — устанавливаемые атрибуты (см. листинг 1.42);
- 5-й параметр — адрес переменной, которая получит старый атрибут первой из страниц (если их несколько).

В программе встречается еще одна незнакомая нам функция. Это — `FlushInstructionCache`. Она нужна, чтобы очистить буфер, содержащий команды. Если этого не сделать, вполне вероятно, что процессор будет использовать для выполнения старые команды, "не заметив" изменения в памяти. Вот параметры этой функции:

- 1-й параметр — дескриптор процесса, память которого мы меняем;
- 2-й параметр — адрес области, которую мы изменили;
- 3-й параметр — размер изменяемой области.

* * *

И на этом я закончу обсуждение вопроса о модификации исполняемого кода. Не забывайте об этой возможности, когда приступаете к анализу кода.



Глава 2

Инструментарий исследователя машинного кода

Данная глава посвящена различным программным инструментам, которые используются при исследовании и исправлении исполняемых модулей.

2.1. Краткий обзор инструментов

Дадим краткий обзор инструментов, наиболее часто применяемых при исследовании кода, а также рассмотрим несколько простых примеров использования этих инструментов.

Некоторые программы, о которых пойдет речь, созданы энтузиастами-одиночками[1] и живут недолго. Моя задача — не столько описать эти программы, сколько указать, какие инструменты для исследования исполняемого кода существуют, и что от них можно получить. Принципы использования таких программ во многом совпадают. Например, все отладчики реализуют такой инструмент исследования кода, как точка останова. Познакомившись с принципами работы одного отладчика, вы достаточно легко сможете использовать в своей работе и другие подобные программы.

2.1.1. Дизассемблеры

Программа dumpbin.exe

Программа dumpbin.exe входит в состав пакета Visual Studio .NET и используется для исследования загружаемых и объектных модулей COFF-формата, выводя информацию в текущую консоль. Разумеется, консольный вывод

---

[1] Я не использую в книге термин "хакер", поскольку его употребляют в самых разных значениях. Хакером называют и программиста высокого класса, и преступника, который подбором паролей снимает деньги со счетов своей жертвы, часто не имея никакого представления о программировании.

всегда можно перенаправить в текстовый файл, получив, таким образом, возможность подробно изучить дизассемблированный текст. Несмотря на свою консольную природу, данная программа работает довольно толково и вполне годится для анализа небольших программ.

Ключи программы:

- /ALL — выводит всю доступную информацию о модуле, кроме ассемблерного кода;
- /ARCH — выводит содержимое секции .arch заголовка модуля;
- /ARCHIVEMEMBERS — выводит минимальную информацию об элементах объектной библиотеки;
- /DEPENDENTS — выводит имена динамических библиотек, откуда модулем импортируются функции;
- /DIRECTIVES — выводит содержимое секции .drectve, создаваемой компилятором (только для объектных модулей);
- /DISASM — дизассемблирует содержимое секций модуля с использованием символьной (отладочной) информации, если она присутствует;
- /EXPORTS — выводит экспортируемые модулем имена;
- /FPO — выдает на консоль информацию о FPO (frame pointer optimization, оптимизация указателя стека) оптимизации;
- /HEADER — выдает на консоль заголовки модуля и всех его секций. В случае объектной библиотеки выдает заголовки составляющих ее объектных модулей;
- /IMPORTS — выводит имена, импортируемые данным модулем;
- /LINENUMBERS — выдает на консоль номера строк объектного модуля, если таковые имеются;
- /LOADCONFIG — программа выводит структуру IMAGE_LOAD_CONFIG_DIRECTORY, которая используется загрузчиком и которая определена в файле WINNT.H;
- /LINKERMEMBER[:{1|2}] — выводит все имена в объектной библиотеке, определяемые как public;
  - /LINKERMEMBER:1 — в порядке следования объектных модулей в библиотеке;
  - /LINKERMEMBER:2 — вначале выдает смещение и индекс объектных модулей, а затем список имен в алфавитном порядке для каждого модуля;
  - /LINKERMEMBER — сочетание ключей 1 и 2;
- /OUT — определяет, что вывод осуществляется не в консоль, а в файл (например, /OUT:ED.TXT). Конечно, перенаправить вывод в файл можно, просто используя знак >;

- `/PDATA` — выводит содержимое таблиц исключения (для RISC-процессоров);
- `/RAWDATA` — выдает дамп каждой секции файла. Разновидности данного ключа: `/RAWDATA:BYTE`, `/RAWDATA:SHORTS`, `/RAWDATA:LONGS`, `/RAWDATA:NONE`, `/RAWDATA:,`*`number`*. Здесь *`number`* определяет ширину строк;
- `/RELOCATIONS` — выводит все перемещения в таблице перемещений;
- `/SECTION:`*`section`* — определяет конкретную анализируемую секцию;
- `/SUMMARY` — выдает минимальную информацию о секциях;
- `/SYMBOLS` — выдает таблицу символов COFF-файла.

Пример использования:

```
dumpbin  /disasm  prog.exe  >prog.txt
```

В текстовый файл prog.txt будет выведен дизассемблированный код программы.

Особенностью программы dumpbin.exe является то, что она дизассемблирует только секции с известными ей именами *(см. разд. 1.5.3)*. Если вы поместите исполняемый код в секцию с произвольным (не предопределенным именем), то программа не будет выводить дизассемблированный код, хотя дамп и выведет.

В качестве интересного примера рассмотрим исследование с помощью программы dumpbin.exe таблицы перемещений *(см. разд. 1.5.3)* динамической библиотеки. Для примера я взял очень простую динамическую библиотеку, написанную на ассемблере. Пусть название библиотеки prog.dll. Выполним команду

```
dumpbin /disasm prog.dll
```

Ниже представлены строки, являющиеся дизассемблированным текстом исполняемого кода модуля. Для некоторых строк я дописал также свой комментарий.

```
10001000: B8 01 00 00 00     mov    eax,1     ;начало процедуры входа
10001005: C2 0C 00           ret    0Ch
10001008: 55                 push   ebp       ;начало экспортируемой
                                              ;функции
10001009: 8B EC              mov    ebp,esp
1000100B: 83 7D 08 01        cmp    dword ptr [ebp+8],1
1000100F: 75 13              jne    10001024
10001011: 6A 00              push   0
10001013: 68 26 30 00 10     push   10003026h
10001018: 68 3E 30 00 10     push   1000303Eh
```

```
1000101D: 6A 00             push   0
1000101F: E8 04 00 00 00    call   10001028 ;вызов функции API
10001024: 5D                pop    ebp
10001025: C2 04 00          ret    4
10001028: FF 25 00 20 00 10 jmp    dword ptr ds:[10002000h]
```

А теперь выведем таблицу перемещения и выясним, в каких командах будут подправляться адреса при загрузке данной динамической библиотеки. Для этого выполним команду

```
dumpbin /relocations prog.dll
```

Вот результат выполнения команды:

```
BASE RELOCATIONS #4
       1000 RVA,        10 SizeOfBlock
       14   HIGHLOW             10003026
       19   HIGHLOW             1000303E
       2A   HIGHLOW             10002000
```

Интересен в первую очередь левый столбец, содержащий смещение операнда, который должен быть учтен при загрузке динамической библиотеки в память. Например, смещение 14 означает, очевидно, адрес `10001014`, т. е. мы попадаем на команду `push 10003026h`. Операнд этой команды, таким образом, представляет адрес, который должен быть откорректирован, если динамическая библиотека будет загружаться по базовому адресу, отличному от `10000000h`.

Знаменитый дизассемблер IDA Pro

Этот знаменитый и никем не превзойденный дизассемблер будет рассмотрен нами в *главе 5*. На момент написания данной книги существует уже версия 4.8 этого продукта. За неимением этой версии я рассматриваю версию 4.7, появившуюся полгода назад. Впрочем, судя по информации, которая опубликована на сайтах **http://www.idapro.ru** и **http://www.idapro.com**, различия в версиях несущественны, во всяком случае, с точки зрения задач, которые стоят перед вашим покорным слугой. Замечу кстати, что IDA Pro является также и отладчиком, но поскольку функции дизассемблирования все же основные, мы и далее будем говорить об этой программе как о дизассемблере.

Дизассемблер W32Dasm

Данному дизассемблеру будет посвящен отдельный *разд. 2.2* — скромнее, чем об IDA Pro. Эта программа, обладающая, как и IDA Pro, возможностями отладки, по-видимому, больше не разрабатывается. Во всяком случае,

версия 10, которую мы и будем рассматривать, создавалась уже, судя по всему, не авторами проекта. В Интернете вы также можете встретить тоже весьма хорошую версию 8.98.

Специализированные дизассемблеры

Что я понимаю под специализированными дизассемблерами? Это дизассемблеры, ориентированные на определенные компиляторы. Речь идет не о декомпиляторах. Перевод исполняемого кода в исходный текст программы, т. е. декомпиляция, в общем случае невозможен. Специализированные дизассемблеры распознают структуры языка — классы, события, методы и др., и дизассемблируют их. Чаще всего в этой связи упоминается Delphi, т. к. анализ программ, написанных на этом языке с помощью обычного дизассемблера, весьма затруднителен. Единственной программой, известной автору, которая неплохо справляется с программами, написанными на языках Delphi и C Builder, является дизассемблер DeDe (по-видимому, это Delphi Decompiler). Сайт создателя данного дизассемблера располагается по адресу **http://dafixer.cjb.net/**.

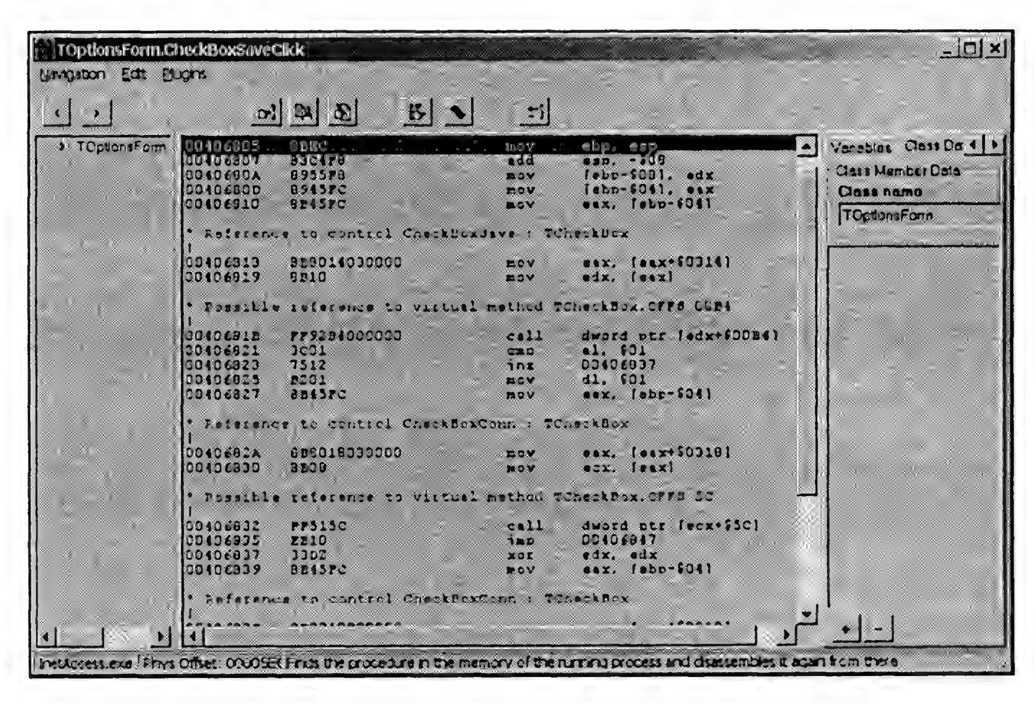


Рис. 2.1. Окно программа DeDe.exe, представляющее дизассемблированный текст события нажатия одной из кнопок окна исследуемого приложения

С помощью программы DeDe в течение нескольких минут вы получите полное представление об иерархии объектов программы, а главное — сможете просмотреть ассемблерный код любого события, например, нажатие кнопки, или события, связанного с созданием формы. На рис. 2.1 вы можете видеть одно из окон программы, которое содержит дизассемблированный текст события нажатия кнопки.

2.1.2. Отладчики

Отладчики — это программы, позволяющие в пошаговом режиме выполнять программы в машинном коде. Все известные системы программирования, как правило, имеют свои встроенные отладчики, мы о них говорить не будем. Наша задача — рассмотреть независимые, если так можно сказать, инструменты отладки. Большинство современных отладчиков понимают структуру отладочной информации основных компиляторов, если таковая присутствует в отлаживаемом модуле. В этом случае они способны отлаживать программу как на уровне ассемблерного кода, так и на уровне текста программ, что, конечно, облегчает анализ. Однако такая ситуация встречается далеко не часто — кому надо оставлять в рабочем пакете отладочную информацию, разве только начинающему программисту.

Еще одна особенность современных отладчиков — это появление в них элементов, свойственных дизассемблирующим программам: распознавание библиотечных и API-функций, возможность корректировать текст и писать комментарии[2]. Налицо, таким образом, сближение дизассемблеров и отладчиков. Так как ранее мы говорили, что и многие дизассемблеры имеют возможность выполнять модуль в режиме отладки, то сближение, как видим, происходит с двух сторон. Как в последствии мы неоднократно убедимся, что наиболее эффективным является исследование исполняемого кода при параллельном использовании и отладчика, и дизассемблера.

Turbo Debugger

Turbo Debugger — один из самых популярных отладчиков 1990-х годов и в настоящее время, к сожалению, фирмой Borland не поддерживается. Версия же середины 90-х годов прошлого века, которая свободно распространяется в Интернете, весьма неустойчиво работает в операционных системах Windows NT/2000/XP/Server 2003[3]. Между тем, для отладки небольших

---

[2] Впрочем, отладка ведь осуществляется уже над ассемблерным текстом, так что начальное дизассемблирование любой отладчик и так осуществляет.

[3] Вот что можно прочесть на сервере фирмы Borland: "The Turbo Debugger is provided "as is," without warranty of any kind. Borland does not offer technical support or accept bug reports on this version and it will not be updated or upgraded. The latest, supported version of our debugger is available in Borland C++Builder".

простых приложений и в учебных целях этот отладчик вполне можно использовать (рис. 2.2).

Рис. 2.2. Окно отладчика Turbo Debugger с загруженной в него программой

Debugging Tools for Windows

Программы Debugging Tools for Windows входит в пакет "The Microsoft® Windows® Driver Development Kit (DDK) for Windows XP". Некоторое время назад этот пакет можно было свободно скачать непосредственно с сайта фирмы Microsoft: **http://www.microsoft.com**. Данный инструментарий позволяет отлаживать как обычные приложения, так и драйверы режима ядра. В пакет входит программа windbg.exe (версия 6.0), имеющая графический интерфейс. В нашу задачу не входит изучение данного отладчика. Замечу только, что при отладке основным является событийный механизм: вы ставите точки прерывания (останова, breakpoints) на выполнение какой-либо функции API или обращении к какой-либо области памяти загруженного отладчиком приложения, а далее запускаете отладку и работаете в данном приложении, пока работа не прервется обращением к отслеживаемой функции или области памяти. После этого вы находите в отладчике код, который осуществил обращение к функции или памяти, и анализируете этот фрагмент. В общем, такой работой мы будем заниматься в *главе 4*, когда подробно будем говорить об отладчике SoftICE.

6 Зак. 119

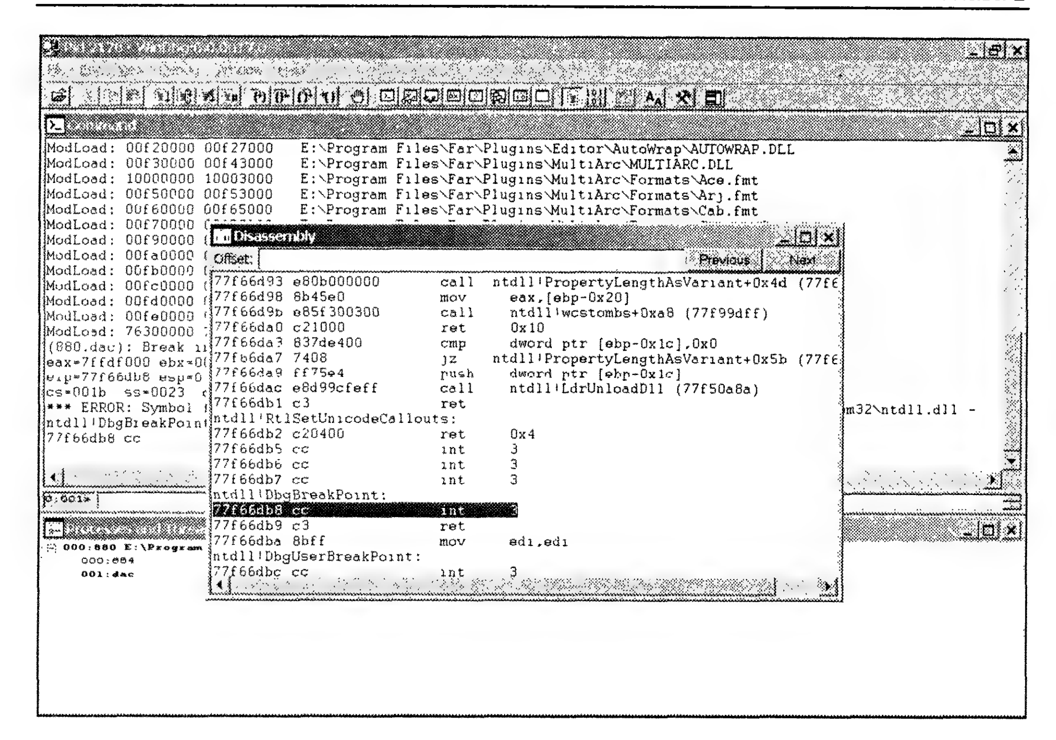


Рис. 2.3. Графический интерфейс программы windbg.exe

Отладчик OllyDbg

Отладчик OllyDbg один из лучших в настоящее время отладчиков прикладных программ. Подробнее о нем будем говорить в *разд. 2.3*. Англоязычный форум, посвященный данному отладчику, можно найти по адресу **http://ollydbg.win32asmcommunity.net/stuph/**. Официальный сайт (адрес **http://home.t-online.de/home/Ollydbg**) автора программы уже некоторое время не функционирует. Имеется также сайт **http://www.ollydbg.de/**.

Мощный отладчик SoftICE

Да, великий и ужасный SoftICE. Разработчиком данного продукта является фирма NuMega Lab. В 1997 г. она была куплена корпорацией Compuware. Название SoftICE неслучайно. Дело в том, что когда активизируется данный отладчик, все программное обеспечение компьютера "замораживается". Вы получаете мгновенный снимок со всей системы. *Глава 4* будет полностью посвящена описанию этого замечательного продукта.

Информацию об отладчике, а также других продуктах NuMega Lab можно найти по адресу **http://www.compuware.com/products/numega.htm**.

2.1.3. HEX-редакторы

Что это такое HEX-редактор? HEX (от англ. *hexadecimal*) означает шестнадцатеричный, т. е. это редакторы, работающие с шестнадцатеричными числами, точнее, данными в шестнадцатеричном формате. Чаще всего речь идет о редактировании файлов, но это также могут быть и области диска. Однако более продвинутые программы обладают возможностью дизассемблировать двоичный код и делать исправления указанием мнемонических имен команд.

WinHex

Программа обладает поистине фейерверком возможностей:

- может работать с файлами и различает при этом множество форматов, выполняет также различные операции над файлами: шифрование, сравнение, разбивку и объединение файлов и многое другое;
- может работать с диском на низком уровне; программа незаменима при восстановлении потерянных файлов;
- может редактировать файлы, расположенные непосредственно в памяти.

Сайт создателей данной программы расположен по адресу **http://www.x-ways.net/**. Однако возможности дизассемблирования у программы отсутствуют, что несколько снижает ее шансы в исследовании кода.

Hacker Viewer

Данная программа (hiew.exe) широко известна в среде программистов, занимающихся исследованием, да и исправлением исполняемого кода. Название программы происходит от фразы "Hacker's view". Основная задача, которую выполняет данная программа, — просмотр и редактирование загружаемых модулей. Причем просмотр и редактирование допускаются в трех вариантах: двоичный, текстовый и ассемблерный.

Программа имеет консольный интерфейс (рис. 2.4). Все команды выполняются при помощи функциональных клавиш (в том числе в сочетании с клавишами <Alt> или <Ctrl>). Например, нажимая клавишу <F4>, вы получаете возможность выбрать способ представления двоичного файла: текстовый, ассемблерный или двоичный. Нажимая клавишу <F3> (при условии, если вы находитесь в двоичном или ассемблерном просмотре), получаете возможность редактировать файл. Если же, находясь в ассемблерном просмотре, вы после нажатия клавиши <F3> нажмете еще и клавишу <F2>, то сможете редактировать машинную команду в символьном виде. Мы не будем далее останавливаться на командах данной программы, поскольку они просты, очевидны и могут быть получены по нажатию клавиши <F1>, а перейдем сразу к простому примеру использования данной программы.

```
Hiew: dexem.exe
   dexem.exe        FRO   NE 00000272 a16 --------        25099 Hiew 6.81 (c)SEN
 0000025F: C05E5F3F                 rcr         b,[bp][5F],03F ;"?"
 00000263: E55D                     in          ax,05D
 00000265: C3                       retn
 00000266: 90                       nop
 00000267: B806BC                   mov         ax,0BC06 ;"╝"
 0000026A: 014740                   add         [bx][40],ax
 0000026D: 1C25                     sbb         al,025 ;"%"
 0000026F: 4A                       dec         dx
 00000270: 5E                       pop         si
 00000271: 5A                       pop         dx
 00000272: AE                       scasb
 00000273: 16                       push        ss
 00000274: 132D                     adc         bp,[di]
 00000276: 6E                       outsb
 00000277: 98                       cbw
 00000278: 97                       xchg        di,ax
 00000279: 5E                       pop         si
 0000027A: 025F04                   add         bl,[bx][04]
 0000027D: EB18                     jmps        000000297  -----  (1)
 0000027F: 43                       inc         bx
 00000280: 42                       inc         dx
 00000281: B737                     mov         bh,037 ;"7"
 00000283: 80FF3C                   cmp         bh,03C ;"<"
```

Рис. 2.4. Интерфейс программы hiew.exe

В листинге 2.1 представлена простая консольная программа на языке ассемблера, выводящая на экран текстовую строку.

Листинг 2.1

```
.586P
.MODEL FLAT, stdcall
;константы
STD_OUTPUT_HANDLE equ -11
INVALID_HANDLE_VALUE  equ -1
;прототипы внешних процедур
EXTERN  GetStdHandle@4:NEAR
EXTERN  WriteConsoleA@20:NEAR
EXTERN  ExitProcess@4:NEAR
;директивы компоновщику для подключения библиотек
includelib f:\masm32\lib\user32.lib
includelib f:\masm32\lib\kernel32.lib
;-------------------------------------------------
;сегмент данных
_DATA SEGMENT
       BUF   DB    "Строка для вывода",0
       LENS  DWORD ?     ;количество выведенных символов
```

```
       HANDL DWORD ?
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
;получить HANDLE вывода
       PUSH STD_OUTPUT_HANDLE
       CALL GetStdHandle@4
       CMP  EAX,INVALID_HANDLE_VALUE
       JNE  _EX
       MOV  HANDL,EAX
;вывод строки
       PUSH 0
       PUSH OFFSET LENS
       PUSH 17
       PUSH OFFSET BUF
       PUSH HANDL
       CALL WriteConsoleA@20
_EX:
       PUSH 0
       CALL ExitProcess@4
_TEXT ENDS
END START
```

Программа из листинга 2.1 проста и корректна. Представьте теперь, что при отладке вы случайно изменили одну команду: вместо `JE` поставили `JNE`. В результате после трансляции программа перестала работать. Можно исправить ее, не прибегая к ассемблерному тексту? Конечно. Для этого вначале ее следует дизассемблировать, найти ошибку, а потом воспользоваться программой hiew.exe. Вообще говоря, можно ограничиться только программой hiew.exe, т. к. она вполне прилично дизассемблирует небольшие программы. Однако мы нарочно проведем исправление в два этапа.

Дизассемблируем модуль при помощи программы dumpbin.exe. В листинге 2.2 дизассемблированный текст программы.

Листинг 2.2

```
Dump of file cons1.exe
File Type: EXECUTABLE IMAGE
```

```
00401000: 6AF5           push  0F5h
00401002: E82B000000     call  00401032
00401007: 83F8FF         cmp   eax,0FFh
0040100A: 751E           jne   0040102A
0040100C: A316304000     mov   [00403016],eax
00401011: 6A00           push  0
00401013: 6812304000     push  403012h
00401018: 6A11           push  11h
0040101A: 6800304000     push  403000h
0040101F: FF3516304000   push  dword ptr ds:[00403016h]
00401025: E80E000000     call  00401038
0040102A: 6A00           push  0
0040102C: E80D000000     call  0040103E
00401031: CC             int   3
00401032: FF2508204000   jmp   dword ptr ds:[00402008h]
00401038: FF2500204000   jmp   dword ptr ds:[00402000h]
0040103E: FF2504204000   jmp   dword ptr ds:[00402004h]
```

По дизассемблированному коду легко обнаружить ошибку. Кстати, команду

```
cmp eax, 0FFFFFFFFh
```

надо, естественно, понимать как

```
cmp eax, -1
```

Запомним нужный код 83F8FFH. Запускаем программу hiew.exe, нажимаем клавишу <F7> и ищем нужное сочетание. Далее нажимаем клавишу <F3>, затем клавишу <F2> и после заменяем команду JNE на JE. Клавиша <F9> фиксирует изменение. В результате мы исправили программу без ее повторной трансляции. Конечно, нужную команду можно найти и по адресу 00401007H, поскольку программа hiew.exe корректно отображает виртуальные адреса дизассемблируемых секций.

Автором программы, которая, кстати, различает кроме формата PE еще и другие форматы исполняемых файлов (MZ, NE, LX, LE, ELF), является Евгений Сусликов. Сайт поддержки программы: **http://www.serje.net/sen/**.

Biew.exe

Эта программа по своему внешнему виду и системе команд очень близка к программе hiew.exe. Программа также поддерживает большое число форматов исполняемых файлов. Сайт поддержки **http://biew.sourceforge.net**. Последняя распространяемая версия — 5.62.

2.1.4. Другие утилиты

Существует огромное количество всевозможных программ, помогающих понять структуру исполняемого модуля и исполняемый код, который в нем находится. Все они выполняют ту или иную функцию. Скажем, PE-браузеры разрешают получить наиболее полную информацию об исполняемом модуле. Примером такого простейшего браузера может служить программа из *приложения*. Другие более совершенные браузеры позволяют редактировать PE-заголовок — исправлять содержимое полей, добавлять новые секции и т. д. О двух других типах исследовательского инструментария будет сказано в следующих разделах. Мы не будем называть конкретные программы, т. к. во-первых, их много, а во-вторых, большинство таких программ долго не поддерживаются авторами.

Исследователи ресурсов

Программ, которые могут просматривать ресурсы исполняемого модуля, имеется огромное количество. Программ, которые могут вытаскивать ресурсы и сохранять в двоичном виде или формате текстового (RC) файла, меньше. Более продвинутые программы позволяют редактировать ресурсы непосредственно в самом исполняемом модуле.

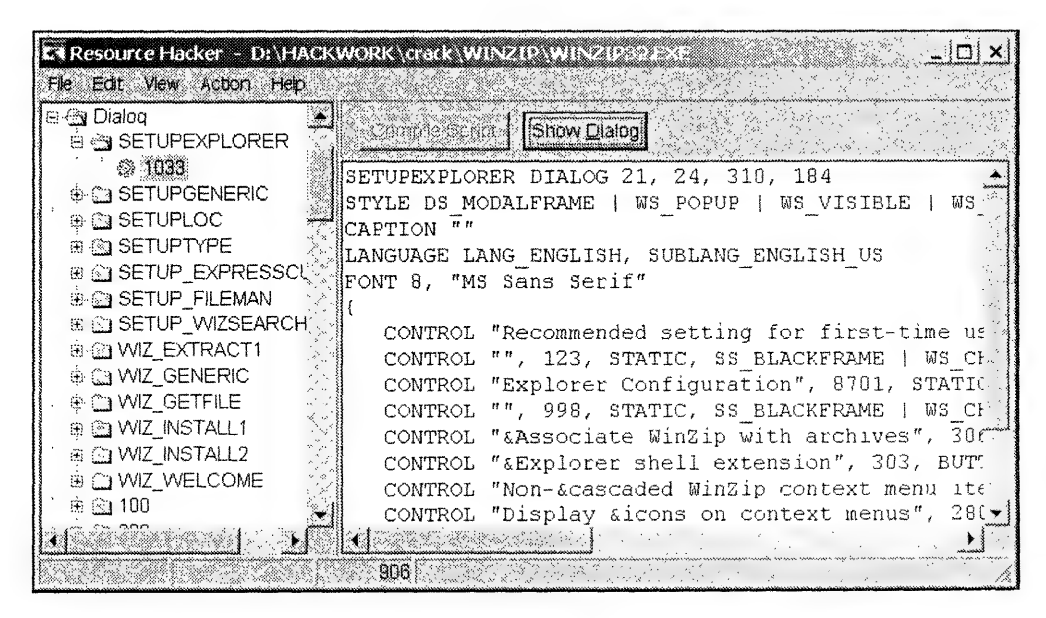


Рис. 2.5. Окно одной из программ, позволяющей исследовать исполняемый модуль

На рис. 2.5 показана одна из программ исследования ресурсов исполняемых модулей. Обратите внимание на левую панель, где перечислены все ресурсы данной программы в виде иерархической структуры. Правая панель окна содержит текст ресурса в формате RC. Вы можете исправить текст прямо здесь, а далее с помощью кнопки **Compile Script** откомпилировать текст и поместить его в модуль. Но этим возможности программы не заканчиваются. Посредством кнопки **Show Dialog** мы можем вызвать редактор диалоговых окон и отредактировать окно с помощью визуальных средств, а затем поместить полученный ресурс опять в исполняемый модуль.

Мониторы

Мониторы — это особый вид программ для отслеживания определенных событий в системе. По отношению к нашим с вами задачам монитор отслеживает определенные действия программ. Существуют два вида мониторов, которые нам наиболее интересны. Первые отслеживают обращение программ к системному реестру, вторые — к файловой системе вообще.

На рис. 2.6 представлено окно монитора, отслеживающего обращение прикладных программ к системному реестру. Имея такой протокол, мы легко можем определить, что делала та или иная программа с содержимым реестра. Узнав это, мы получаем важный ключ поиска нужного места в исполняемом коде. Аналогично работают файловые мониторы.

Registry Monitor - Sysinternals: www.sysinternals.com

File Edit Options Help

#	Time	Process	Request	Path	Result
25951	38 99137500	wmplayer exe 3560	QueryValue	HKCU\Software\Microsoft\MediaPlayer\Preferences\TitlebarLeaveTime	NOTFOUNC
25952	39 00941026	wmplayer exe 3560	QueryValue	HKCU\Software\Microsoft\MediaPlayer\Preferences\CDRecordFileDetail	NOTFOUNC
25953	39 00944958	wmplayer exe 3560	QueryValue	HKLM\SOFTWARE\Microsoft\MediaPlayer\Preferences\CDRecordFileDe	NOTFOUNC
25954	39 00948857	wmplayer exe 3560	QueryValue	HKCU\Software\Microsoft\MediaPlayer\Preferences\CDRecordFileSepara	NOTFOUNC
25955	39 00951687	wmplayer exe 3560	QueryValue	HKLM\SOFTWARE\Microsoft\MediaPlayer\Preferences\CDRecordFileSe	NOTFOUNC
25956	39 00955557	wmplayer exe 3560	QueryValue	HKCU\Software\Microsoft\MediaPlayer\Preferences\CDRecordPath	NOTFOUNC
25957	39 00958335	wmplayer exe 3560	QueryValue	HKLM\SOFTWARE\Microsoft\MediaPlayer\Preferences\CDRecordPath	NOTFOUNC
25958	39 00966350	wmplayer exe 3560	OpenKey	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\ShellCompatibility	NOTFOUNC
25959	39 00994868	wmplayer exe 3560	OpenKey	HKCU\Software\Policies\Microsoft\WindowsMediaPlayer\onlinePages\	NOTFOUNC
25960	39 01000790	wmplayer exe 3560	QueryValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Ena	NOTFOUNC
25961	39 01007888	wmplayer exe 3560	QueryValue	HKCU\Software\Microsoft\MediaPlayer\Preferences\AutoMetadataUpdare	NOTFOUNC
25962	39 01011829	wmplayer exe 3560	QueryValue	HKCU\Software\Microsoft\MediaPlayer\Preferences\AutoMetadataFuzzy	NOTFOUNC
25963	39 01015448	wmplayer exe 3560	QueryValue	HKCU\Software\Microsoft\MediaPlayer\Preferences\AutoOrganize	NOTFOUNC
25964	39 01018588	wmplayer exe 3560	QueryValue	HKCU\Software\Microsoft\MediaPlayer\Preferences\MetadataRetrieval	SUCCESS
25965	39 01022209	wmplayer exe 3560	OpenKey	HKCU\Software\Policies\Microsoft\WindowsMediaPlayer\	NOTFOUNC
25966	39 01025640	wmplayer exe 3560	OpenKey	HKCU\Software\Policies\Microsoft\WindowsMediaPlayer\	NOTFOUNC

Рис. 2.6. Окно программы-монитора, отслеживающей все обращение прикладных программ к системному реестру

Существуют и другие программы, работающие с системным реестром. Такие программы следовало бы назвать *сканерами*. Они могут определять, к каким

данным реестра был доступ в течение конкретного промежутка времени. Довольно часто такие программы более удобны, нежели мониторы.

2.2. Дизассемблер и отладчик W32Dasm

Программа W32Dasm (Windows Disassembler) представляет собой симбиоз довольно мощного дизассемблера и отладчика. Версии программы 8.93 и 10 — наиболее распространены в настоящее время и могут работать не только с PE-модулями, но и с DOS-, NE-, LE-модулями. Я намерен довольно полно описать работу с этой программой.

2.2.1. Начало работы

Интерфейс и настройки программы

Внешний вид программы показан на рис. 2.7. Меню дополняется панелью инструментов, элементы которой активизируются в зависимости от ситуации.

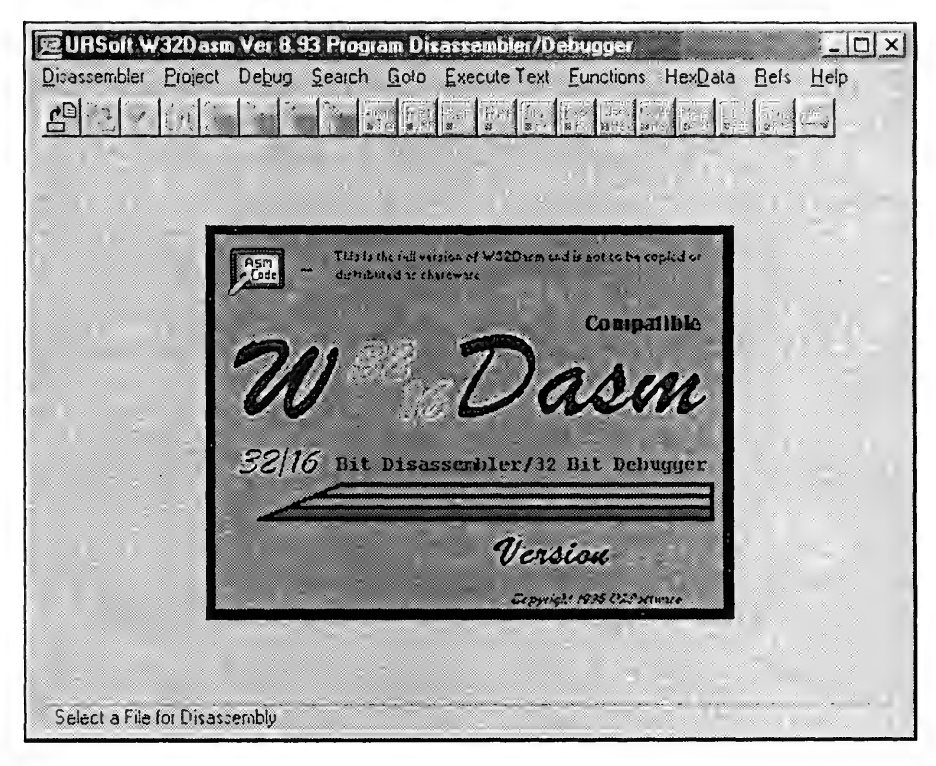


Рис. 2.7. Интерфейс программы W32Dasm

Как уже было сказано, программа является дизассемблером и отладчиком в одном лице. Это отражено также в двух пунктах меню: **Disassembler** и **Debug**. Соответственно, имеются отдельные настройки для дизассемблера и отладчика. Для дизассемблера существуют всего три опции, касающиеся анализа перекрестных ссылок в условных переходах, безусловных переходах и вызовах процедур. По умолчанию все три опции установлены. Отмена этих опций нежелательна, т. к. снижает информативность дизассемблированного текста. В принципе, отмена указанных опций может понадобиться при дизассемблировании очень большой программы, чтобы несколько ускорить процесс анализа кода программы.

Опций отладчика несколько больше, но они все очевидны. Окно установки опций отладчика изображено на рис. 2.8, все они касаются особенностей загрузки процессов, потоков и динамических библиотек.

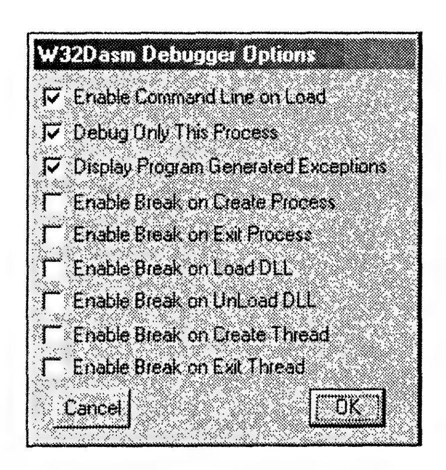


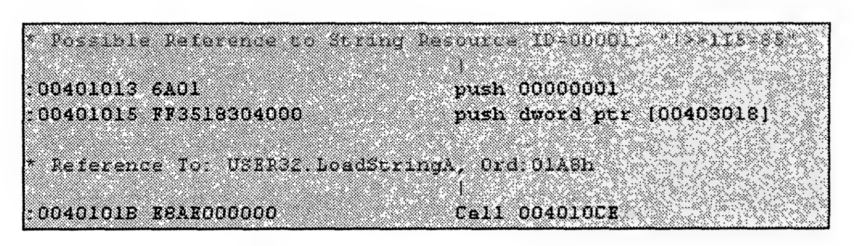
Рис. 2.8. Опции отладчика

Для начала работы с исполняемым модулем достаточно выбрать нужный файл в меню **Disassembler | Open File**. После этого программа производит анализ модуля и выдает дизассемблированный текст, а также весьма полную информацию о секциях модуля[4]. W32Dasm весьма корректно распознает API-функции и комментирует их (рис. 2.9).

После работы с модулем можно создать проект работы при помощи пункта **Disassembler | Save Disassembler**. По умолчанию проект сохраняется в подкаталог wpjfiles, который расположен в рабочем каталоге W32Dasm, и состоит из двух файлов: с расширением alf — дизассемблированный текст, с расши-

[4] Хотя W32Dasm работает с разного типа модулями, я буду рассматривать только модули формата PE.

рением wpj — собственно сам проект. При повторном запуске можно открывать уже не модуль, а проект с помощью пункта **Project | Open**.



```
* Possible Reference to String Resource ID=00001: "!>>1I$-$$"
                                          |
:00401013 6A01                            push 00000001
:00401015 FF3518304000                    push dword ptr [00403018]

* Reference To: USER32.LoadStringA, Ord:01A8h
                                          |
:0040101B E8AE000000                      Call 004010CE
```

Рис. 2.9. Фрагмент дизассемблированного текста

2.2.2. Работа с дизассемблируемым кодом

Перемещение по дизассемблированному тексту

При перемещении по тексту текущая строка подсвечивается другим цветом, при этом особо выделяются переходы и вызовы процедур. Передвижение облегчается также с помощью пунктов меню **Goto**:

- **Goto Code Start** — переход на начало листинга;
- **Goto Program Entry Point** — переход на точку входа программы, наиболее важная команда меню;
- **Goto Page** — переход на страницу с заданным номером, по умолчанию число строк на странице равно пятидесяти;
- **Goto Code Location** — переход по заданному адресу, в случае отсутствия адреса учитываются диапазон и близость к другим адресам.

Другой способ передвижения по дизассемблированному тексту предоставляет меню **Search** — поиск. Здесь нет никаких отличий от подобных команд других программ.

В случае, если текущая строка находится в команде перехода или вызова процедуры, с помощью кнопок, расположенных на панели инструментов, можно перейти по соответствующему адресу. Такое передвижение можно продолжать, пока вы не обнаружите нужный фрагмент программы. Но самое приятное здесь то, что можно передвигаться и в обратном направлении. При этом нужные кнопки на панели инструментов автоматически подсвечиваются.

Кроме того, адреса, куда производится переход, содержат список адресов, откуда производятся переходы. Подсветив строку, где расположен адрес, и дважды щелкнув правой кнопкой мыши по этому адресу, мы перейдем к соответствующей строке.

Отображение данных

Есть несколько вариантов работы с данными.

Во-первых, имеется пункт меню **HexData | Hex Display of Data**, где можно просмотреть содержимое сегментов данных в шестнадцатеричном и строковом варианте. Кроме того, сам код программы также можно просматривать в шестнадцатеричном виде. Для этого используется пункт **HexData | Hex Display of Code**.

Во-вторых, существует пункт меню **Refs | String Data References**. Это весьма мощное и полезное средство. При выборе этого пункта появляется список строк, на которые имеются ссылки в тексте программы. Во всяком случае это то, что сумел определить дизассемблер при анализе программы. Выбрав нужную строку, можно двойным щелчком перенестись в соответствующее место программы. Если ссылок на данную строку несколько, то, продолжая делать двойные щелчки, мы будем переходить во все нужные места программы. На рис. 2.10 изображено окно ссылок на строковые типы данных.

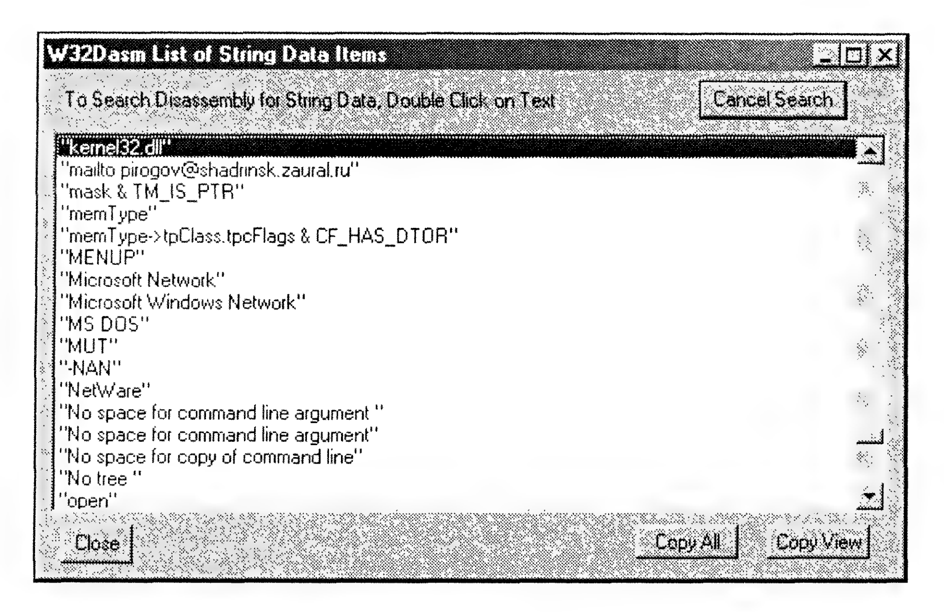


Рис. 2.10. Окно ссылок на строки

Как видно из рисунка, можно скопировать в буфер выбранную строку или все строки.

Вывод импортированных и экспортированных функций

Список импортированных функций и модулей находится в начале дизассемблированного текста (рис. 2.11). Кроме того, список импортированных функций можно получить из меню **Functions | Imports**. Выбрав нужную функцию в списке, двойным щелчком можно получить все места программы, где вызывается эта функция.

```
+++++++++++++++++++ IMPORTED FUNCTIONS ++++++++++++++++++
Number of Imported Modules =    7 (decimal)

   Import Module 001: ADVAPI32.dll
   Import Module 002: KERNEL32.dll
   Import Module 003: MPR.dll
   Import Module 004: COMCTL32.dll
   Import Module 005: GDI32.dll
   Import Module 006: SHELL32.dll
   Import Module 007: USER32.dll

+++++++++++++++++++ IMPORT MODULE DETAILS +++++++++++++++

   Import Module 001: ADVAPI32.dll

 Addr:000D9660 hint(0000) Name: RegCloseKey
 Addr:000D966E hint(0000) Name: RegOpenKeyExA
 Addr:000D967E hint(0000) Name: RegQueryValueExA
 Addr:000D9692 hint(0000) Name: RegSetValueExA

   Import Module 002: KERNEL32.dll
```

Рис. 2.11. Фрагмент списка импортированных модулей и функций

Экспортированные функции также можно получить в соответствующем окне, выбрав пункт **Functions | Exports**.

Отображение ресурсов

В начале дизассемблированного текста описаны и ресурсы, точнее два основных ресурса — меню и диалоговое окно. Со списком этих ресурсов можно работать и в специальных окнах, получаемых с помощью пунктов меню программы **Refs | Menu References** и **Refs | Dialog References**. Строковые ресурсы можно увидеть в уже упомянутом окне просмотра перечня строковых ссылок (см. рис. 2.10). Остальные ресурсы данной версии программы, к сожалению, не выделяются.

Операции с текстом

Строки дизассемблированного текста могут быть выделены и скопированы в буфер либо напечатаны. Выделение строки осуществляется щелчком левой кнопки мыши, когда курсор мыши расположен в крайнем левом положении. Для выделения группы строк дополнительно следует нажать клавишу <Shift>. Выделенный фрагмент копируется специальной кнопкой, которая "загорается", когда фрагмент существует, либо отправляется на печатающее устройство.

2.2.3. Отладка программ

Кратко рассмотрим возможности отладки программы W32Dasm.

Загрузка программ для отладки

Загрузить модуль для отладки можно двумя способами. С помощью пункта **Debug | Load Process** загружается для отладки уже дизассемблированный модуль.

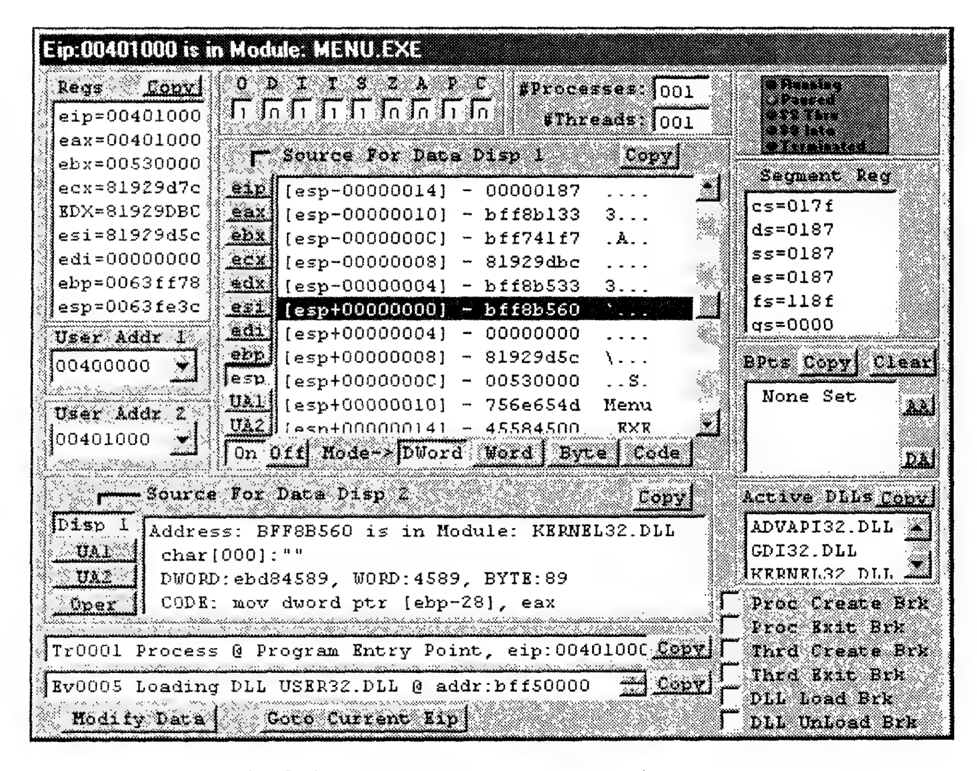


Рис. 2.12. Первое окно отладчика — информационное

Пункт **Debug | Attach to an Active Process** позволяет "подсоединяться" и отлаживать процесс, находящийся в памяти. После загрузки отладчика на экране появляются два окна. Первое окно — информационное (рис. 2.12), в документации оно называется "нижним левым окном отладчика". Второе окно — управляющее (рис. 2.13), называемое в документации "нижним правым окном отладчика".

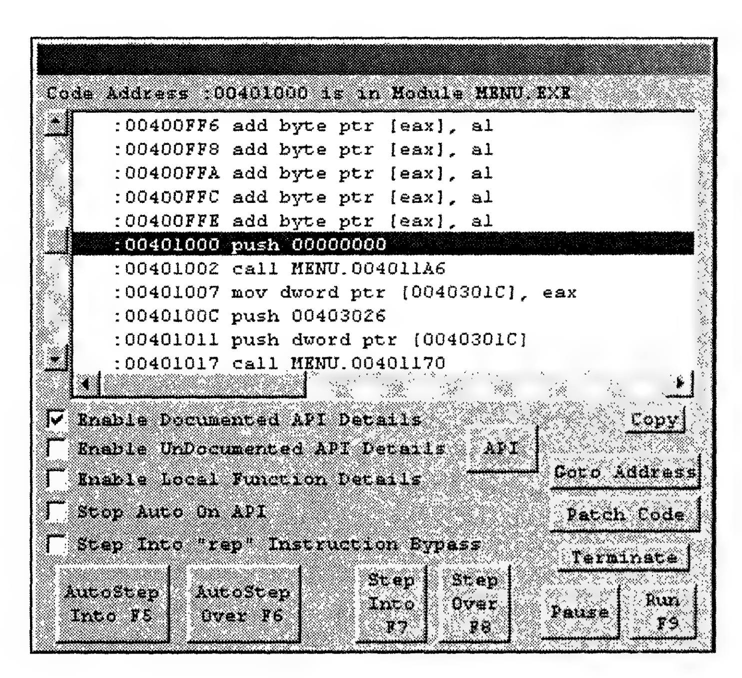


Рис. 2.13. Второе окно отладчика — управляющее

Информационное окно содержит несколько окон-списков: содержимое регистров микропроцессора, значения флагов микропроцессора, точки останова, содержимое сегментных регистров, историю трассировок, историю событий, базовые адреса, два дисплея данных. Далее я объясню также значения кнопок этого окна.

Обратимся теперь к управляющему окну. Кнопка **Run F9** запускает загруженную в отладчик программу, кнопка **Pause** приостанавливает работу программы, кнопка **Terminate** останавливает выполнение программы и выгружает ее из отладчика. Кнопки **Step Over F8** и **Step Into F7** используются для пошагового исполнения программы. Первая кнопка, выполняя инструкции, "перескакивает" код процедур и цепочечные команды с повторением, вторая кнопка выполняет все инструкции последовательно. Кроме того, имеются кнопки **AutoStep Over F6** и **AutoStep Into F5** для автоматического пошагового выполнения программы. В случае API-функций даже нажатие кнопки

Step Into F7 не приведет к пошаговому выполнению кода функции в силу того, что код функции не доступен для пользовательских программ. Очень удобно, что при пошаговом выполнении происходит передвижение не только в окне отладчика, но и в окне дизассемблера.

Отмечу, что если вы подсоединяетесь к процессу, расположенному в памяти, то при выходе из отладчика процесс также будет выгружен из памяти, что может привести к неправильной работе операционной системы.

Работа с динамическими библиотеками

Для отладки динамической библиотеки можно поступить следующим образом. Загрузить в отладчик программу, которая обращается к динамической библиотеке. Затем обратиться к списку используемых динамических библиотек. Возможно, для работы с данной динамической библиотекой вам понадобится запустить программу и выполнить какую-либо ее функцию. Дважды щелкнув по нужной библиотеке, вы получите дизассемблированный код данной библиотеки в окне дизассемблера и возможность работать с кодом библиотеки.

Точки останова

В дизассемблированном тексте можно установить *точки останова*. Для этого следует перейти к нужной строке и воспользоваться клавишей <F2> или левой кнопкой мыши при нажатой клавише <Ctrl>. Установка точки останова в окне дизассемблера тут же отражается в информационном и управляющем окнах — у отмеченной команды появляется префикс BP. Удалить точку останова можно тем же способом, что и при установке. Точку останова можно сделать также неактивной. Для этого нужно обратиться к информационному окну и списку точек останова. Выбрав нужный адрес, щелкните по нему правой кнопкой мыши. При этом "звездочка" у данной точки останова исчезнет, а строка в окне дизассемблера из желтой станет зеленой.

Быстрый переход к точке останова можно произвести, выбрав ее из списка (информационное окно) и сделав двойной щелчок мышью. Наконец, можно установить точки останова на определенные события, такие как загрузка и выгрузка динамической библиотеки, создание и удаление потока и т. д. Все это делается при помощи установки соответствующих флажков в информационном окне.

Модификация кода, данных и регистров

Отладчик позволяет модифицировать загруженный в него код (рис. 2.14). Сделать это можно, обратившись к кнопке **Patch Code** в управляющем окне (см. рис. 2.13). Важно отметить, что модификации подвергается только код, загруженный в отладчик, а не дизассемблированный текст. Найдя нужное

место в отлаживаемом коде и модифицировав его, вы можете тут же проверить результат модификации, запустив программу. Если модификация оказалась правильной, можно приступать уже к модификации самого модуля.

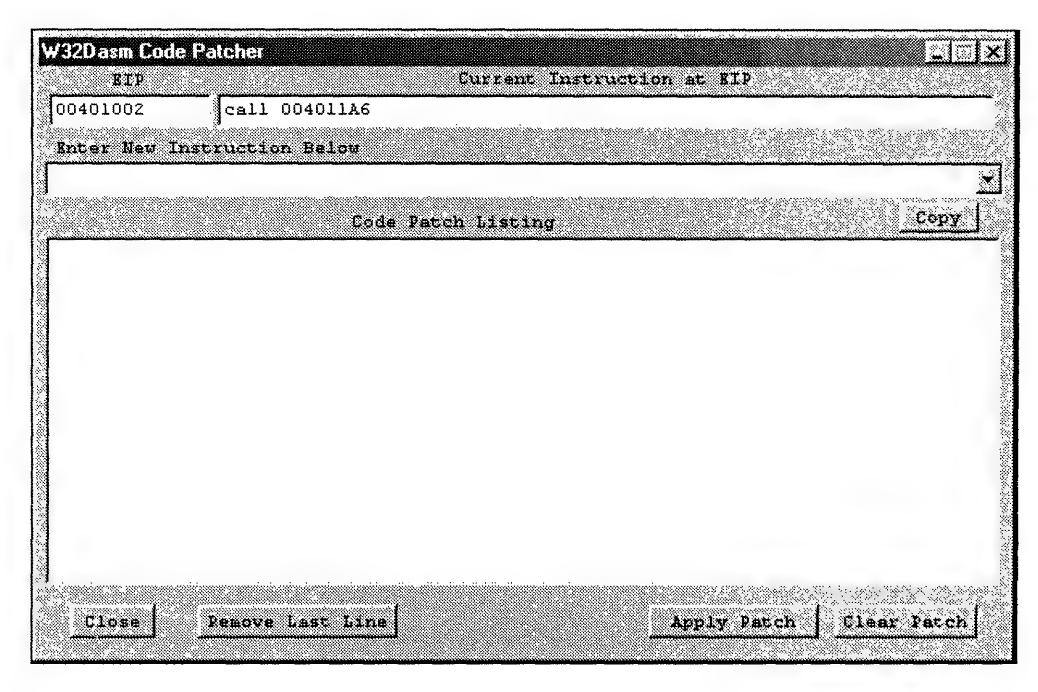


Рис. 2.14. Окно модификации отлаживаемого кода

Для модификации регистров и ячеек памяти исполняемого процесса существует специальная кнопка **Modify Data** в информационном окне (см. рис. 2.12). Окно модификации изображено на рис. 2.15. Окно несколько загромождено элементами, но, присмотревшись, вы поймете, что все элементы на своем месте. В верхней части окна расположены текущие значения основных флагов микропроцессора, которые вы можете изменить. Для того чтобы модифицировать содержимое регистра или ячейку памяти, следует вначале установить модифицирующую величину — **Enter Value**. Далее следует выбрать нужный регистр (в списке регистров, см. рис. 2.15) и нажать кнопку слева от имени этого регистра в списке. Чтобы установить старое значение, следует нажать кнопку **R** справа от регистра. Чтобы изменить содержимое ячейки памяти, нужно вначале записать адрес ячейки в поле раскрывающегося списка **Mem Lock** (рис. 2.15), а затем воспользоваться кнопкой **Mem**. Другие операции, предоставляемые данным окном, также достаточно очевидны.

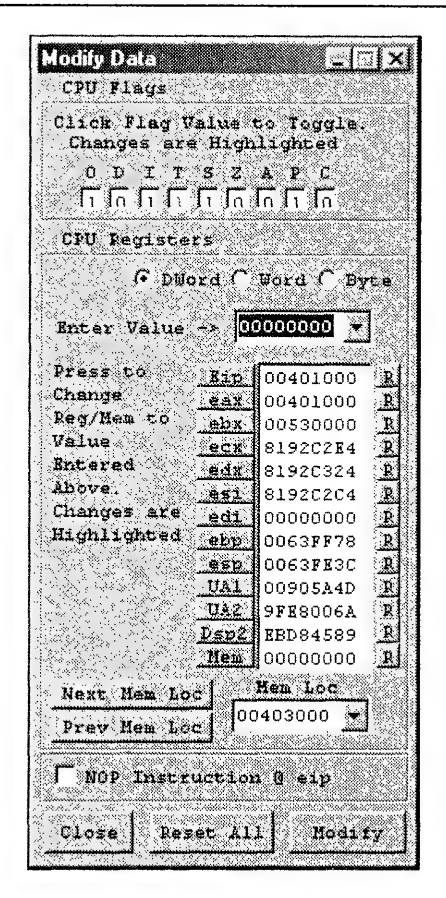


Рис. 2.15. Окно модификации регистров и ячеек памяти

Дополнительные возможности для работы с API

Отладчик позволяет выдавать дополнительную информацию о выполняемых API-функциях. Чтобы воспользоваться этим, необходимо сделать следующее. В управляющем окне установите флажки: **Enable Documented API Details**, **Stop Auto On API** (см. рис. 2.13). Далее запустите программу на выполнение нажатием клавиши <F5>. При прохождении API-функции будет производиться остановка, а на экране — появляться окно с информацией о данной функции.

Поиск нужного места в программе

Часто требуется найти в дизассемблированном коде место, соответствующее месту исполняемой программы. Наиболее эффективно это можно сделать следующим образом. Загружаем в отладчик данный модуль. Запускаем его,

доходим до нужного места и нажимаем кнопку **Terminate**. В результате подсвеченная строка в дизассемблированном коде окажется как раз в нужном месте. Нужно только иметь в виду, что некоторые программы делают изменения, которые потом продолжают действовать и после прерывания ее работы. К таковым относятся, в частности, горячие клавиши.

К использованию программы W32Dasm мы еще вернемся в последующих главах.

2.3. Отладчик OllyDbg

Это удивительный отладчик. Например, он умеет определять параметры процедуры, циклы, выделять константы, массивы и строки, что никогда не было отличительным признаком такого рода инструментов. Отладчик поддерживает все процессоры 80*x*86 и знает множество числовых форматов. Можно загружать в отладчик исполняемый модуль или подключаться к уже работающему процессу. В общем, возможностей — море, и мы будем о некоторых из них говорить.

2.3.1. Начало работы с отладчиком

Окна отладчика

Начнем рассмотрение отладчика OllyDbg с изучения главного окна этой программы (рис. 2.16). Кроме естественного горизонтального меню и панели кнопок, в главном окне расположены по умолчанию четыре информационных окна: окно дизассемблера (левое верхнее), окно данных (левое нижнее), окно регистров (правое верхнее), окно стека (правое нижнее). Кроме указанных окон в процессе работы можно использовать и другие окна. Перечень всех информационных окон представлен в пункте меню **View**. С частью окон вы познакомитесь в процессе изучения данного раздела, о других вы можете узнать самостоятельно, если конечно будете использовать данный инструмент, что я вам настоятельно рекомендую.

Обратимся теперь к окнам, которые мы видим на рис. 2.16. Это наиболее важные окна, без которых никак не обойтись в процессе отладки.

Окно дизассемблера

Окно состоит из четырех колонок.

- Колонка адреса команды (**Address**). В данной колонке показан виртуальный адрес команды, который она получает при загрузке модуля в память. Двойной щелчок мышью в данной колонке переводит все адреса в смещения относительно текущего адреса (`$`, `$-2`, `$+4` и т. п.).

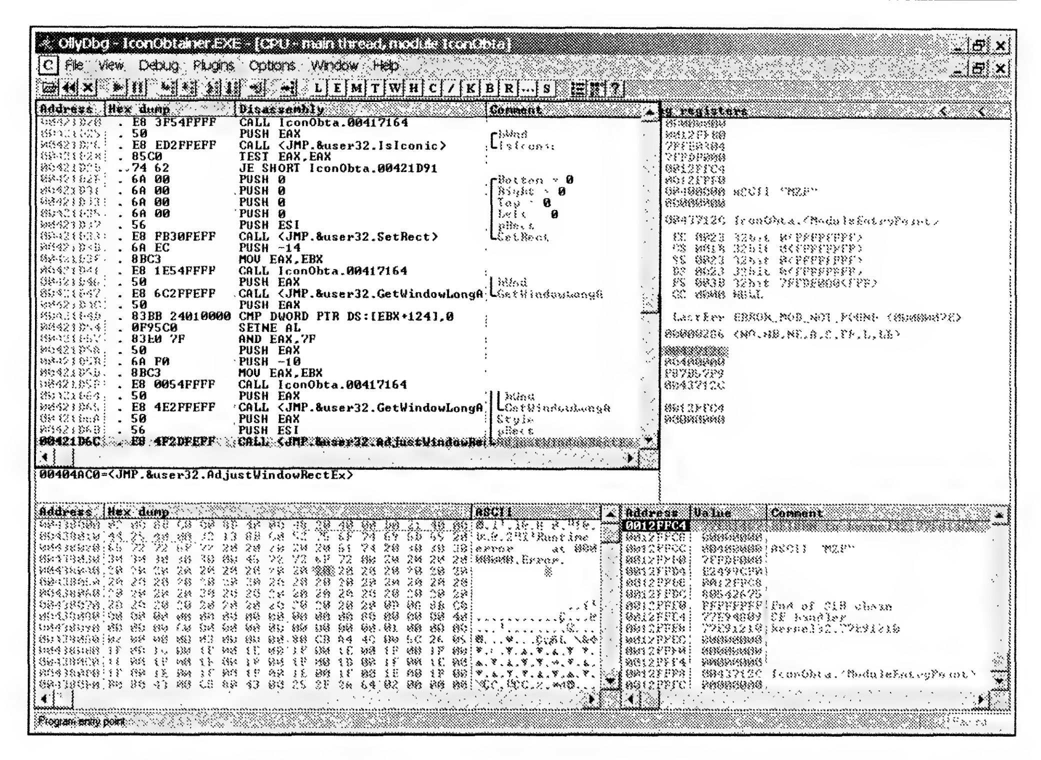


Рис. 2.16. Отладчик OllyDbg
с загруженной в него программой

- Колонка кода команды (**Hex dump**). При этом выделяются собственно код и значение операнда. Кроме этого, в колонке имеются различные значки, которые помогают разобраться в логике программы: указывают на команду, на которую есть переходы (>), команду, осуществляющую переход (^ — вверх, ˅ — вниз), и т. п. В этой же колонке отмечаются циклы, которые удалось распознать программе. Двойной щелчок по этой колонке приводит к тому, что в первой колонке адрес будет подсвечен красным. Это означает, что мы установили точку останова на данную команду (адрес).
- Колонка команды (**Disassembly**). В этой колонке представлено ассемблерное обозначение команды. Двойной щелчок по колонке приводит к тому, что появляется окно редактирования ассемблерной команды. Вы можете исправить команду, и далее в отладке будет участвовать исправленная вами команда. Более того, вы можете записать исправленный текст программы в исполняемый модуль. Здорово, не правда ли?
- Колонка комментария (**Comment**). Здесь программа помещает дополнительную информацию о команде. В частности, указываются имена API-

функций, библиотечных функций и т. д. Сделав двойной щелчок по этой колонке, мы получим возможность добавлять свой комментарий к каждой строке ассемблерного кода.

Окно данных

Окно имеет по умолчанию три колонки: колонка адреса (**Address**), колонка, содержащая шестнадцатеричное значение ячейки (**Hex dump**), колонка текстовой интерпретации содержимого ячеек (**ASCII**, **Unicode** и т. п.). Можно менять смысл второй и третьей колонок. Например, можно интерпретировать содержимое ячеек в кодировке Unicode.

Окно регистров

Окно регистров может содержать три возможных набора: стандартные регистры и регистры сопроцессора, стандартные регистры и регистры MMX, стандартные регистры и регистры в технологии 3DNow. Двойной щелчок в этом окне позволяет редактировать содержимое соответствующего регистра.

Окно стека

Окно стека представляет содержимое стека. Первая колонка (**Address**) содержит адрес ячейки в стеке, вторая колонка (**Value**) — содержимое ячейки, третья колонка (**Comment**) — возможный комментарий к содержимому (см. рис. 2.16).

Еще об окнах

Приступая к работе с отладчиком, имейте в виду следующее.

- Щелкнув правой кнопкой мыши по любому из окон, вы получите контекстное меню. Оно индивидуально для каждого из четырех окон. Советую подробно изучить эти меню. Часть информации вы можете получить в процессе нашего изложения.
- Окна (их содержимое) не являются независимыми. Посмотрите на регистры. Щелкнув правой кнопкой мыши по одному из рабочих регистров, можно всегда перевести его содержимое, как адрес в области данных (follow in dump) или в области стека (follow in stack).

Отладочное выполнение

Отладка — это анализ поведения программы путем исполнения ее в различных режимах. Вот о различных режимах выполнения программы в отладчике OllyDbg мы сейчас и поговорим.

Итак, исполняемый код загружен в отладчик. В окне дизассемблера мы видим ассемблерный код.

Какие же основные способы выполнения программы можно использовать?

- Пошаговое выполнение с обходом процедур (step over). При нажатии клавиши <F8> выполняется текущая ассемблерная команда. Выполняя одну команду за другой, мы можем в трех остальных окнах следить за тем, как меняется содержимое регистров, секции данных и секции стека. Особенностью данной команды является то, что если очередной командой будет команда вызова процедуры (CALL), то автоматически будут выполняться все команды процедуры (все команды процедуры выполнятся как одна инструкция).
- Пошаговое выполнение с заходом в процедуру (step into). Выполнение осуществляется по нажатию клавиши <F7>. Основным отличием от предыдущего способа является то, что при встрече с командой CALL далее пошагово будут выполняться инструкции процедуры.
- Оба способа пошагового выполнения (step over и step into) можно автоматизировать, если использовать так называемую *анимацию* (animation), соответственно, нажимая комбинации клавиш <Ctrl>+<F8> или <Ctrl>+<F7>. При нажатии этих комбинаций клавиш команды "step over" и "step into" будут выполняться в автоматическом режиме одна за другой с небольшой задержкой. После каждой инструкции окна отладчика будут обновляться, так что можно отслеживать динамику изменений. В любой момент можно приостановить выполнение, нажав клавишу <Esc>. Выполнение приостанавливается также на точках останова *(см. разд. 2.3.2)* и в случае, если исполняемая программа генерирует исключение.
- Еще один способ пошагового выполнения программы — это *трассировка* (trace). Она напоминает анимацию, но при этом на каждом шаге не обновляются окна отладчика. Два способа трассировки, соответствующие "step over" и "step into", выполняются с помощью комбинаций клавиш <Ctrl>+<F12> и <Ctrl>+<F11>. Остановить трассировку можно теми же способами, что и анимацию. После каждой команды информация о ее выполнении заносится в специальный трассировочный буфер, который можно просмотреть с помощью пункта меню **View | Run trace**. При желании содержимое буфера можно сбросить в текстовый файл. Можно определить условия, по которым будет происходить остановка трассировки (Set trace condition) — через комбинацию клавиш <Ctrl>+<T> (точка останова). При этом можно задать:
  - диапазон адресов, в котором будет произведен останов;
  - условные выражения, например, EAX>100000, при выполнении которых трассировка будет остановлена;
  - номер команды или набор команд, по которым будет произведен останов.

Можно заставить отладчик выполнить код, пока не встретится возврат из процедуры (execute till return). Другими словами, будет выполнен весь код текущей процедуры и осуществлен возврат из нее. Для этого предназначена комбинация клавиш <Ctrl>+<F9>.

Наконец, если в процессе трассировки вы оказались глубоко в системном коде, можно дать команду выхода из него (execute till user code) — нажать комбинацию клавиш <Alt>+<F9>.

2.3.2. Точки останова

Точки останова (точки прерывания, контрольные точки) — это очень мощное средство отладки приложения. Они позволяют разобраться в логике выполнения программы, давая мгновенные снимки регистров, стека и данных в определенные моменты выполнения.

Обычные точки останова

Обычные точки останова (ordinary breakpoints) ставятся на конкретную команду. Для этого в окне дизассемблера используется клавиша <F2> или двойной щелчок мыши во второй колонке окна кода (**Hex dump**). В результате адрес команды в первой колонке (**Address**) окрашивается по умолчанию в красный цвет. Этот вид точек останова, в первую очередь, помогает найти корреляцию между наблюдаемым нами ходом выполнения программы (появление окон, сообщений и т. п.) и конкретными участками программного кода. Кроме этого, в точке останова можно проверить состояние регистров, переменных, состояние стека. Вторичное нажатие клавиши <F2> в точке останова или двойной щелчок мыши удаляют точку останова. Имейте в виду, что остановка осуществляется перед выполнением "помеченной" команды.

Условные точки останова

Условные точки останова (conditional breakpoints) устанавливаются по нажатию комбинации клавиш <Shift>+<F2>. При этом появляется окно с комбинированным списком, куда можно занести точку останова. В поле комбинированного списка задается условие, при выполнении которого должна быть произведена остановка на данной команде. Отладчик поддерживает достаточно сложные выражения, содержащие условия. Приведу несколько примеров:

- `EAX==1` — остановка на отмеченной команде (перед ее выполнением) будет осуществлена, если содержимое регистра `EAX` будет равно `1`;

- EAX=0 AND ECX>10 — остановка на отмеченной команде будет осуществлена, если содержимое регистра EAX будет равно 0, а содержимое регистра ECX будет больше 10;
- [STRING 427010]=="Error" — в данном случае выполнение программы приостановится, если по адресу 427010H будет располагаться строка "Error". Можно написать и так: EAX=="Error", и тогда содержимое EAX будет трактоваться как указатель на строку;
- [427070]=1231 — данное условие определяет остановку, если содержимое ячейки памяти 427070H равно 1231H;
- [[427070]]=1231 — здесь используется косвенная адресация. Предполагается, что ячейка с адресом 427070H содержит адрес другой ячейки, содержимое которой и будет сравниваться с числом 1231H.

Условные точки останова с записью в журнал

Данный вид точек останова (conditional logging breakpoint) является расширением условных точек останова. Устанавливается по нажатию комбинации клавиш <Shift>+<F4>. Каждый раз, когда данная точка останова срабатывает, делается запись в журнале. Посмотреть содержимое журнала можно, нажав комбинацию клавиш <Alt>+<L> или выбрав из меню **View | Log**. Можно установить запись, которая станет появляться в журнале, а также указать выражение, значение которого будет записываться в журнал. Наконец, можно установить счетчик, который будет показывать, сколько раз должна быть произведена запись в журнал и нужно ли прерывать работу программы каждый раз, когда выполняются условия останова.

Точка останова на сообщения Windows

Поскольку сообщения приходят на функцию окна (точнее класса окна), то для установки точки останова на сообщение необходимо наличие окон; другими словами, оконное приложение должно быть запущено. Итак, для простоты я загрузил в отладчик простое приложение всего с одним окном и запустил его при помощи комбинации клавиш <Ctrl>+<F8>. Через секунду окно приложения активизировалось. Кстати, обратили внимание, какая часть программы непрерывно выполняется? Правильно, цикл обработки сообщений. Чтобы выйти на функцию окна, нужно вызвать список созданных приложением окон. Это делается при помощи пункта меню **View | Windows**. Результат команды мы видим на рис. 2.17.

Из рис. 2.17 можно узнать дескриптор окна, его название, идентификатор и, главное, адрес процедуры класса (столбец **ClsProc**). Последняя информация дает нам возможность обратиться непосредственно к функции окна

и установить там обычную или условную точку останова. Однако при работе с оконными функциями эффективнее использовать точку останова на сообщение.

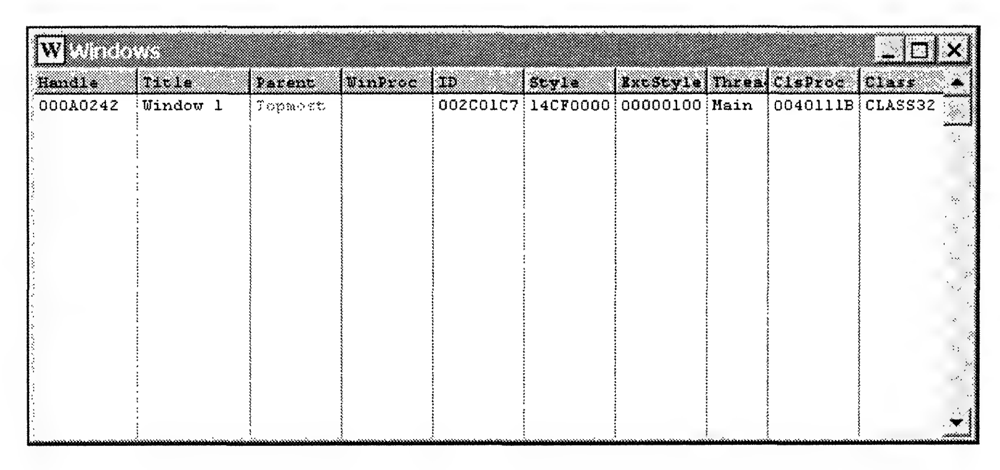


Рис. 2.17. Окно со списком окон, созданных приложением

Итак, щелкнем по окну, изображенному на рис. 2.17, и выберем из контекстного меню пункт **Message breakpoint on ClassProc**. В появившемся окне можно установить параметры точки останова, а именно:

- из выпадающего списка выбрать сообщение. Замечу при этом, что можно выбрать:
  - не само сообщение, а событие, которое может знаменоваться несколькими сообщениями, например, создание и уничтожение окна, событие от клавиатуры и т. п.;
  - сообщения, определяемые пользователем;
- определить перечень окон, которые будут отслеживаться, на предмет поступления данного сообщения: данное окно, все окна с данным заголовком, все окна;
- определить счетчик — сколько раз будет срабатывать точка останова;
- будет или нет останавливаться выполнение программы;
- будет или нет производиться запись в журнал.

Потренируйтесь теперь сами с установкой описанных выше точек останова и проследите также за содержимым окна стека — это весьма поучительное занятие.

Точка останова на функции импорта

Список всех импортируемых с отлаживаемым модулем имен можно получить с помощью нажатия комбинации клавиш <Ctrl>+<N>. Далее, щелкнув правой кнопкой мыши по окну, можно установить:

- точку останова на вызов импортируемой функции (команда **Toggle breakpoint on import**);
- условную точку останова на вызов импортируемой функции (команда **Conditional breakpoint on import**);
- условную точку останова на импорт с записью в журнал (команда **Conditional log breakpoint on import**);
- точки останова на все ссылки на данное имя (команда **Set breakpoint on every reference**);
- точки останова с записью в журнал на все ссылки на данное имя (команда **Set log breakpoint on every reference**)

или удалить все точки останова (команда **Remove all breakpoints**).

Точка останова на область памяти

Отладчик OllyDbg позволяет установить одну *точку останова на область памяти*. Выбираем окно дизассемблера или окно данных (dump). Далее используем контекстное меню и выбираем пункт **Breakpoint | Memory on access** (на доступ к памяти) или **Breakpoint | Memory on write** (на запись в память). После этого точка останова готова к использованию. Как вы понимаете, первый тип точки останова возможен и для данных, и для кода, второй — только для кода. Удалить точку останова на область памяти можно опять же из контекстного меню: **Breakpoint | Remove memory breakpoint**.

Точка останова в окне *Memory*

Окно **Memory** отображает блоки памяти, которые были зарезервированы для отлаживаемой программы или самой отлаживаемой программой. Вот в этом окне также можно установить одну точку останова. Для этого опять используется контекстное меню, появляющееся посредством щелчка правой кнопкой мыши и выбором пункта **Set memory breakpoint on access** (Установить точку останова на доступ к памяти) или **Set memory breakpoint on write** (Установить точку останова на запись в память). Удалить точку останова можно из того контекстного меню командой **Remove memory breakpoint**.

Аппаратные точки останова

Обычные точки останова используют стандартный вектор прерывания `INT 3`. Добавление таких точек останова может существенно замедлить вы-

полнение отлаживаемой программы. Но, как известно, у микропроцессора Intel Pentium имеются четыре отладочных регистра DR0—DR3 *(см. разд. 1.2)*. Эти регистры могут содержать четыре контрольные точки — виртуальные адреса текущей программы. Как только адрес, который использует команда, оказывается равным адресу в одном из указанных регистров, так генерируется исключение, перехватываемое отладчиком. *Аппаратные точки останова* не замедляют выполнение отлаживаемой программы, но, как видно из сказанного выше, их может быть всего 4. Установить аппаратную точку останова можно из окна дизассемблера с помощью пункта **Breakpoint | Hardware on execution** контекстного меню либо в окне данных с помощью пунктов **Breakpoint | Hardware on access** или **Breakpoint | Hardware on access**. Удалить аппаратные точки останова можно с помощью того контекстного меню: **Breakpoint | Remove hardware breakpoints**.

2.3.3. Другие возможности

Окно наблюдения

В отладчике OllyDbg имеется окно для наблюдения за выражениями. С выражениями мы уже сталкивались, когда рассматривали условные точки останова. Вы можете использовать сколь угодно сложные выражения, в которых участвуют ячейки памяти и регистры. Окно наблюдения вызывается командой меню **View | Watches**. Щелкнув в появившемся окне правой кнопкой мыши и выбрав пункт **Add Watches** (Добавить наблюдение), вы можете определить выражение, за которым отладчик будет наблюдать, т. е. выводить значение этого выражения. На рис. 2.18 представлено окно наблюдения, содержащее список из четырех выражений, значения которых отслеживаются при каждом выполнении команды процессора и отображаются в окне.

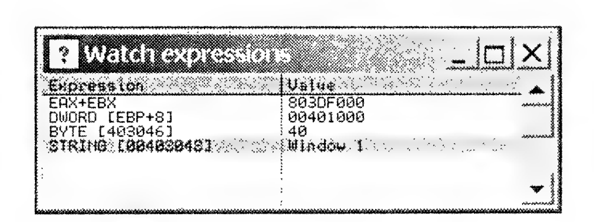


Рис. 2.18. Окно наблюдения за выражениями

Поиск информации

Отладчик OllyDbg позволяет эффективно искать различного рода информацию. Рассмотрим некоторые возможности.

По команде от нажатия комбинации клавиш <Ctrl>+<B> появляется окно поиска, где вы можете определить строку, которая будет разыскиваться в загруженном в отладчик модуле. Строку для поиска можно вводить в виде последовательности символов, байтов, символов в кодировке Unicode.

Для поиска команд используются комбинации клавиш <Ctrl>+<F> для одиночной команды и <Ctrl>+<S> для последовательности команд.

Нажатие комбинации клавиш <Ctrl>+<L> повторяет последний сделанный поиск.

Исправление исполняемого модуля

Отладчик OllyDbg обладает великолепной возможностью записи исправления в исполняемый модуль. Вы можете не только переписать с исправлениями отлаживаемый модуль, но и создать новый исполняемый модуль. Делается это очень просто. Для этого щелкаем правой кнопкой мыши в окне дизассемблера и выбираем пункт **Copy to execution | Selection**. В результате весь дизассемблированный модуль вместе с исправленными командами будет скопирован в новое окно. После этого опять щелкаем по этому окну правой кнопкой мыши и выбираем пункт **Save file**. Далее вы можете выбрать, под каким именем будет сохранен (создан) исполняемый модуль. Это действительно очень удобно: во-первых, вы можете создавать произвольное количество версий исправленного кода, во-вторых, проверка правильности исправление осуществляется, не выходя из отладчика.

На этом я закончу рассмотрение отладчика OllyDbg, хотя остается еще огромное количество интересных вопросов, связанных с использованием этой замечательной программы. Увы, все в этом мире заканчивается, и объем книги требует переходу к следующим вопросам.

2.4. Несколько примеров редактирования исполняемых модулей

Примеры, которые будут здесь рассмотрены, не столь сложны. Я привожу их для того, чтобы:

- во-первых, продемонстрировать возможности тех инструментов, о которых я рассказал выше;
- во-вторых, показать несколько стандартных приемов, которые используются для исследования и исправления исполняемого кода;
- в-третьих, доколе же можно испытывать терпение читателя и кормить его одними обещаниями.

Лишний раз подчеркну, что все примеры исправления исполняемого кода приводятся в данной книге только в учебных целях.

2.4.1. Пример 1. Удаление нежелательного сообщения

История эта приключилась со мной недавно. Я приобрел компакт-диск с исторической энциклопедией. Установив ее на компьютер и проверив, что все работает, я на некоторое время забыл о программе. Спустя неделю я обнаружил, что при запуске программы появляется следующее окно (рис. 2.19).

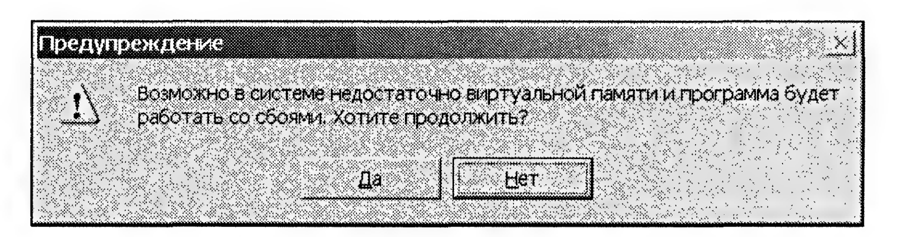


Рис. 2.19. Сообщение, которое мешало мне работать

Если нажать кнопку **Да**, то программа будет работать, причем совершенно нормально. Немного поразмыслив, я пришел к выводу, что причиной появления такого сообщения являлся перенос файла виртуальной памяти в другой раздел, что я сделал дня за два до этого. Поскольку возвращать все в исходное состояние я не собирался, а программа все равно работала нормально, я решил просто исправить исполняемый код и убрать раздражающее меня сообщение.

Итак, приступим. Судя по всему, перед нами простое окно `MessageBox`. Признаком такого окна является, во-первых, пиктограмма в левой части (в данном случае восклицательный знак), а во-вторых, две кнопки — **Да** и **Нет**. Возникает вопрос, как добраться до этого вызова?

Поиск в OllyDbg

Выйти на сообщение `MessageBox` в отладчике можно разным способом. Проще всего установить точку останова на импортируемое имя `MessageBox` *(см. разд. 2.3.2)*, а затем запустить программу (комбинация клавиш <Ctrl>+ +<F8>) и ждать остановки. Но в данном случае все гораздо проще. Сообщение появляется в начале запуска программы, и выйти на это место в программе можно простым пошаговым выполнением кода (клавиша <F8>).

На рис. 2.20 представлено окно отладчика с разыскиваемым нами фрагментом. Обратим внимание, что чуть выше вызова функции `MessageBox` имеется

вызов API-функции `GlobalMemoryStatus`. Очевидно, анализ результата выполнения именно этой функции и приводит к появлению сообщения об ошибке. В данном случае нет смысла пытаться понять, как работает данная функция. Важно, что после вызова этой функции идут следующие строки:

```
00494039  813D 287A4900  CMP DWORD PTR DS:[497A28],989680
00494043  7D 1F          JGE SHORT RHistory.00494064
```

Проще заменить `JGE SHORT 00494064` на `JMP SORT 00494064`. Тем самым мы обойдем вызов окна `MessageBox`.

Мы с вами подробно рассматривали замечательную программу hiew.exe *(см. разд. 2.1.3)*, с помощью которой можно редактировать исполняемые модули. Однако мы знаем, что отредактировать исполняемый модуль можно и с помощью отладчика OllyDbg. Вообще, понимание, что у задачи может быть несколько решений, и вы знаете эти решения, придает уверенности. Желаю вам, дорогой читатель, как можно больше таких незабываемых минут!

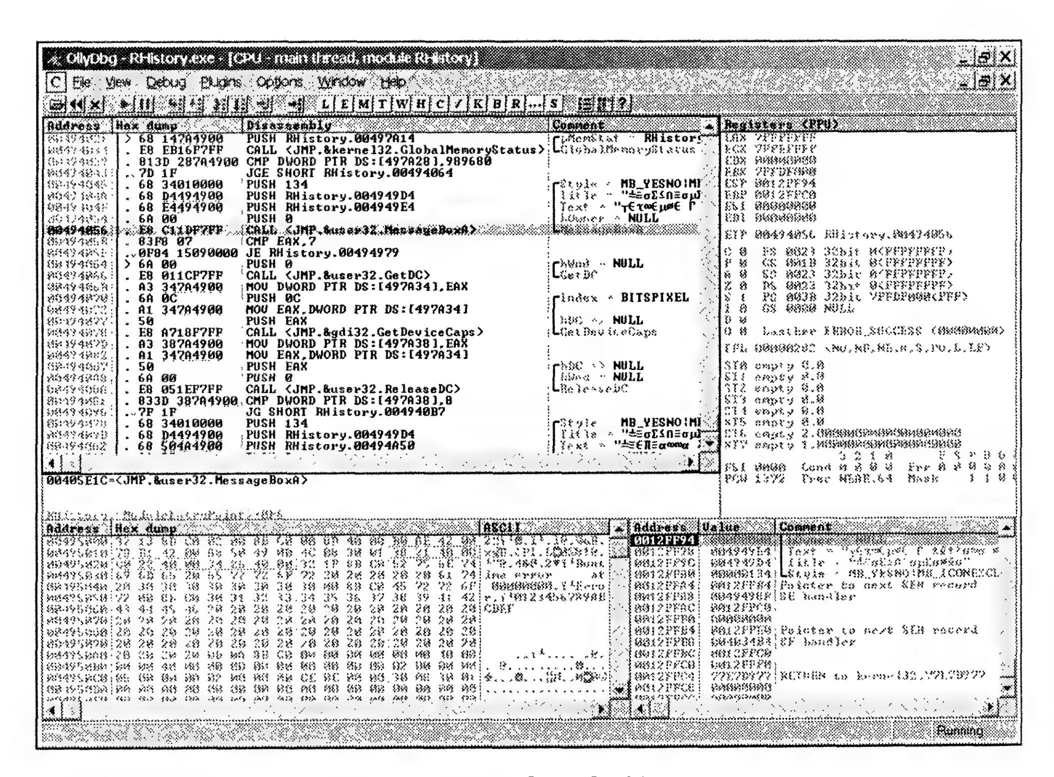


Рис. 2.20. Окно OllyDbg
с фрагментом вызова `MessageBox`

Поиск в W32Dasm

Попробуем теперь поискать нужное нам место в дизассемблере W32Dasm. Здесь можно обратиться к списку импортируемых функций и, разыскав `MessageBox` с помощью двойных щелчков мыши по строке, где записана эта функция, найти все места в программе, где происходит вызов этой функции. Но можно использовать и отладчик, встроенный в W32Dasm, что мы сейчас и сделаем.

Используем пункт меню **Debug | Load Process**. Появится окно для указания параметров загрузки программы. Нажимаем кнопку **Load**. Окно появившегося отладчика изображено на рис. 2.21. Нажимаем на кнопку **AutoStep Over F5** и ждем, когда отобразится так интересующее нас сообщение. Когда сообщение появится, нажимаем кнопку **Terminate** и оказываемся ровно в том месте программы (дизассемблированного текста), куда мы и стремились попасть. Дальнейшие действия нам уже знакомы — запускаем hiew.exe и делаем нужные исправления.

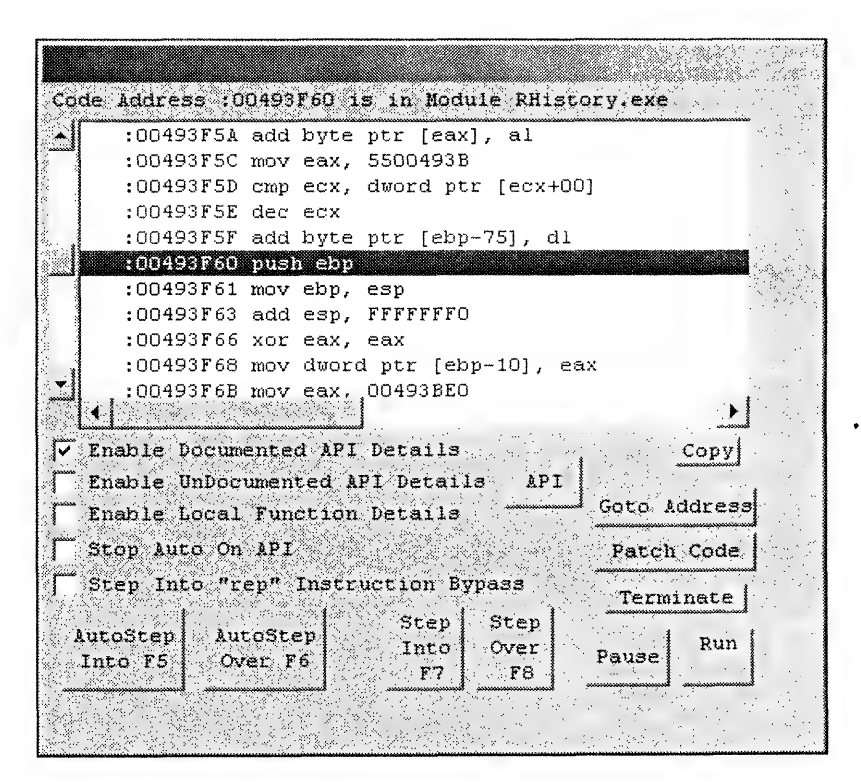


Рис. 2.21. Окно отладчика W32Dasm

Поиск в IDA Pro

Поиск нужного фрагмента в IDA Pro также вполне традиционен. В окне функций найдем `MessageBox` и щелчком выйдем на фрагмент, представленный на рис. 2.22. Это "переходник", который вызывается из других мест программы. Обратите внимание на многоточие. Если щелкнуть по этому месту правой кнопкой мыши и выбрать в контекстном меню пункт **Jump to cross reference**, то появится окно со списком всех адресов программы, откуда вызывается функция `MessageBox`. Теперь не составит труда перебрать все вызовы интересующей нас функции и найти нужное место программы.

```
CODE:00405E1C ; ---------------- S U B R O U T I N E ---------------------------------
CODE:00405E1C
CODE:00405E1C ; Attributes: thunk
CODE:00405E1C
CODE:00405E1C                  public MessageBoxA_0 ; weak
CODE:00405E1C MessageBoxA_0    proc near               ; CODE XREF: sub_408560+99↓p
CODE:00405E1C                                          ; sub_4297F0+43↓p ...
CODE:00405E1C                  jmp     ds:__imp_MessageBoxA_0
CODE:00405E1C MessageBoxA_0    endp
CODE:00405E1C
CODE:00405E1C ; ---------------------------------------------------------------------
```

Рис. 2.22. Фрагмент дизассемблированного текста из IDA Pro

Замечание

При исправлении исполняемого кода, а, как правило, речь идет о снятии защиты, очень часто фигурирует команда условного перехода. Но поскольку команде условного перехода очень часто предшествует команда сравнения `CMP`, то часто слышишь утверждение, что для взлома программы надо знать только одну ассемблерную команду. Конечно, это совсем не так, и в сложных ситуациях приходится глубоко "вгрызаться" в ассемблерный код.

2.4.2. Пример 2. Снятие ограничений на использование программы

Задача, которую ставим перед собой, не так сложна, но достаточно распространена. Решить ее можно, воспользовавшись только дизассемблером W32Dasm. Для исправления, как обычно, мы используем программу hiew.exe.

Данная программа (All Screen 95 PRO — программа, с помощью которой можно "снимать" окна и отдельные части экрана) попала ко мне как shareware release довольно давно. Программа написана на Delphi, но мы увидим, что решить поставленную задачу можно, и не зная, на чем написана про-

грамма. Впрочем, дизассемблер DeDe.exe *(см. разд. 2.1.1)*, который я ранее усиленно хвалил, не смог разобраться в данной программе, по-видимому, по причине использования для трансляции программы старого компилятора.

Итак, при запуске программы Allscreen.exe на экране появляется окно, изображенное на рис. 2.23. Ближе познакомившись с предметом, вы убедитесь, что чаще всего приходится искать место в программе, соответствующее какому-либо визуальному эффекту: открытие окна, закрытие окна, вывод текста и т. п.

При нажатии кнопки **Accept** возникает задержка секунд в шесть (рис. 2.24). Далее программа работает нормально.

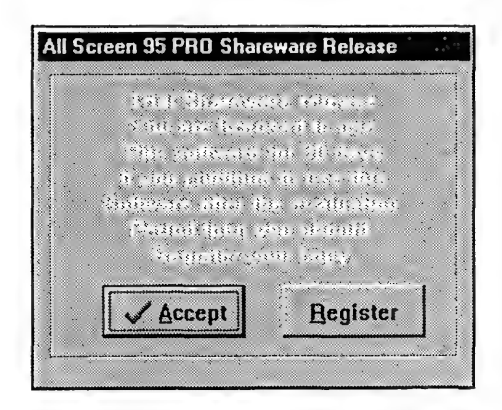


Рис. 2.23. Окно, появляющееся при запуске программы Allscreen



Рис. 2.24. Окно задержки

После пятнадцати запусков появляется окно, представленное на рис. 2.25, и происходит выход из программы.

7 Зак. 119

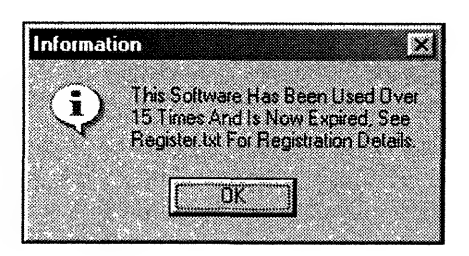


Рис. 2.25. Сообщение об истечении времени работы программы

Таким образом, передо мной стояло две задачи:

- устранить весьма раздражающую меня задержку;
- сделать так, чтобы программа работала при любом количестве запусков.

Процедура задержки

Окно, представленное на рис. 2.25, — это явный "прокол" авторов программы[5]. Дело в том, что окно и все его содержимое можно спрятать в ресурсы. Но когда на том же окне появляется новая запись — это уже программный код. Итак, запускаем дизассемблер W32Dasm и загружаем туда программу Allscreen.exe. Вызываем окно **SDR** (String Data Reference — ссылки на строки данных), ищем строку Shareware Delay, дважды щелкаем по ней и, закрыв его, оказываемся в нужном месте программы. Вот этот фрагмент (листинг 2.3).

Листинг 2.3

```
* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:004420BC(C)
|
:00442123 33D2                  xor  edx, edx
:00442125 8B83B0010000          mov  eax, dword ptr [ebx+000001B0]
:0044212B E8541DFDFF            call 00413E84
:00442130 33D2                  xor  edx, edx
:00442132 8B83B4010000          mov  eax, dword ptr [ebx+000001B4]
:00442138 E8471DFDFF            call 00413E84
:0044213D 33D2                  xor  edx, edx
:0044213F 8B83B8010000          mov  eax, dword ptr [ebx+000001B8]
```

[5] Прокол в том смысле, что данный код был предназначен для защиты, а, следовательно, автор должен был подумать об усложнении преодоления ее.

```
:00442145 E83A1DFDFF        call 00413E84
:0044214A BA50000000        mov  edx, 00000050
:0044214F 8B83BC010000      mov  eax, dword ptr [ebx+000001BC]
:00442155 E8D618FDFF        call 00413A30

* Possible StringData Ref from Code Obj ->"Shareware Delay"
                    |
:0044215A BAA8214400        mov  edx, 004421A8
:0044215F 8B83BC010000      mov  eax, dword ptr [ebx+000001BC]
:00442165 E8EE1DFDFF        call 00413F58
:0044216A 33D2              xor  edx, edx
:0044216C 8B83C0010000      mov  eax, dword ptr [ebx+000001C0]
:00442172 E80D1DFDFF        call 00413E84
:00442177 33D2              xor  edx, edx
:00442179 8B83C4010000      mov  eax, dword ptr [ebx+000001C4]
:0044217F E8001DFDFF        call 00413E84
:00442184 33D2              xor  edx, edx
:00442186 8B83C8010000      mov  eax, dword ptr [ebx+000001C8]
:0044218C E8F31CFDFF        call 00413E84
:00442191 8B83CC010000      mov  eax, dword ptr [ebx+000001CC]
:00442197 E8E8D4FFFF        call 0043F684
:0044219C 5B                pop  ebx
:0044219D C3                ret
```

Я сразу взял чуть больше кода, захватив и несколько верхних строк. По сути дела, перед нами вся процедура задержки. Нет смысла пытаться понять, что означает та или иная команда вызова процедуры `CALL`, хотя легко сообразить (проведя небольшой эксперимент), что, например, `CALL 00413E84` убирает строку с экрана.

Для того чтобы решить проблему задержки, достаточно "выключить" этот фрагмент из программы. Проще всего это можно сделать, поставив в начало фрагмента (адрес `00442123`) команды `POP EBX`/`RET`, используя редактор, как hiew.exe. После запуска исправленной программы задержка действительно исчезает.

Снятие ограничения на количество запусков программы

Перейдем теперь ко второй проблеме — ограничение на количество запусков. Уже из самого вида окна (см. рис. 2.25) ясно, что оно формируется в

самой программе. Следовательно, опять можно попытаться найти текст, который изображается на экране, в самой программе. Как и в предыдущем случае, строка отыскивается в окне **SDR**. Дважды щелкнув по строке, оказываемся в месте программы, представленной в листинге 2.4.

Листинг 2.4

```
:00443326 8BC0              mov  eax, eax
:00443328 53                push ebx
:00443329 8BD8              mov  ebx, eax
:0044332B 803DEC56440001    cmp  byte ptr [004456EC], 01
:00443332 7546              jne  0044337A
:00443334 A124564400        mov  eax, dword ptr [00445624]
:00443339 E84E2CFEFF        call 00425F8C
:0044333E A1D8564400        mov  eax, dword ptr [004456D8]
:00443343 E87816FEFF        call 004249C0
:00443348 FF05F0564400      inc  dword ptr [004456F0]
:0044334E C605EC56440000    mov  byte  ptr [004456EC], 00
:00443355 833DF05644000F    cmp  dword ptr [004456F0], 0000000F
:0044335C 7E1C              jle  0044337A
:0044335E 6A00              push 00000000
:00443360 668B0DB0334400    mov  cx, word ptr [004433B0]
:00443367 B202              mov  dl, 02

* Possible StringData Ref from Code Obj ->"This Software Has Been Used
Over"
                                  |
:00443369 B8BC334400        mov  eax, 004433BC
:0044336E E8BDAEFEFF        call 0042E230
:00443373 8BC3              mov  eax, ebx
:00443375 E84214FEFF        call 004247BC

* Referenced by a (U)nconditional or (C)onditional Jump at Addresses:
|:00443332(C), :0044335C(C)
|
:0044337A 33D2              xor  edx, edx
:0044337C 8B83F4010000      mov  eax, dword ptr [ebx+000001F4]
:00443382 E8A52DFFFF        call 0043612C
:00443387 33D2              xor  edx, edx
```

```
:00443389 8B83F8010000       mov  eax, dword ptr [ebx+000001F8]
:0044338F E8982DFFFF         call 0043612C
:00443394 33D2               xor  edx, edx
:00443396 8B83FC010000       mov  eax, dword ptr [ebx+000001FC]
:0044339C E88B2DFFFF         call 0043612C
:004433A1 33D2               xor  edx, edx
:004433A3 8B8314020000       mov  eax, dword ptr [ebx+00000214]
:004433A9 E87E2DFFFF         call 0043612C
:004433AE 5B                 pop  ebx
:004433AF C3                 ret
```

Опять мы представляем весь необходимый фрагмент (см. листинг 2.4). Просмотрев текст несколько выше ссылки на искомую строку, легко обнаруживаем "подозрительные" команды:

```
cmp dword ptr [004456F0], 0000000F
jle 0044337A
```

Вспомним, что программа перестает работать как раз после пятнадцати запусков (т. е. 0FH в шестнадцатеричном представлении). Проще всего исправить ситуацию, "забив" фрагмент программы с 0044335EH по 00443375H командами NOP (90H), используя программу hiew.exe. В результате программа начинает работать уже без всяких ограничений на количество запусков.

2.4.3. Пример 3. Разбираемся с "Evaluation copy"

Общие соображения

Следующим нашим примером будет попытка сделать из оценочной версии компилятора C++ фирмы Intel версии 4.5, рассчитанной на тридцатисуточное использование, полноценную программу. Конечно, речь не идет о добавлении какой-то функциональности компилятору, если в оценочной версии ее нет. Нет, я всего лишь привожу пример того, как, в принципе, достаточно просто снимается ограничение на использование программы.

После установки компилятора на компьютере мы находим его в каталоге ...\compiler45\bin. Это программа icl.exe. Запустим программу. На консольный экран будут выведены следующие строки.

```
Intel(R) C/C++ Compiler Version 4.5 00015
Copyright (C) 1985-2000 Intel Corporation. All rights reserved.
Evaluation copy.
Icl: NOTE: This is day 1 of 30 day evaluation  period.
Icl: Command line error: no files specified.
```

Последняя строка вполне понятна, мы не указали в командной строке имя файла, содержащего текст программы на языке С. Если это сделать, то компилятор окажется вполне работоспособным. Однако если перевести часы на месяц вперед, то компилятор перестанет работать, а будет выдавать строку "The evaluation period has expired" ("Оценочный период закончился").

Запустим IDA Pro и попытаемся найти строки, выдаваемые компилятором. Вот что я обнаружил через некоторое время поиска:

```
data:00419C20 aCopyrightC1985 db 'Copyright (C) 1985-2000 Intel Corpora-
tion.  All rights reser'
.data:00419C20                              ; DATA XREF: sub_404574+31?o
.data:00419C20                              ; sub_40C974+21?o
.data:00419C20                        db 'ved.',0Ah
.data:00419C20                        db 'Evaluation copy',0
```

Обратите внимание, что это все одна строка. Момент важный. Во всяком случае, можно предположить, что сообщение "Evaluation copy" не связано в программе с какой-то проверкой ограничения на работу программы. Попытка поискать строку, содержащую слова "This is day", однако, успехом не увенчалась. Впрочем, если бы я и нашел такую строку, то в программе, скорее всего, я бы обнаружил вызов библиотечной функции `puts` или `printf`, и для того чтобы найти строки, где проверяется ограничение на использование программы, пришлось бы подниматься на более высокий уровень. Так что я принимаю решение дальнейший анализ перенести в отладчик.

Поиски в отладчике

Прежде чем запускать отладчик, я нахожу адрес функции `main` в дизассемблере IDA Pro. Этот адрес оказывается равным `00402000H`, так что, запустив отладчик OllyDbg, я сразу ставлю точку останова на этот адрес и начинаю анализ программы именно с нее.

Итак, выйдя на функцию `main`, начнем пошаговое выполнение программы (клавиша <F8>), проверяя после каждого вызова процедуры содержимое консольного окна. Очень скоро выходим на следующий фрагмент.

```
0040204A    E8 71040000     CALL  icl.004024C0
0040204F    0FB6C0          MOVZX EAX,AL
00402052    85C0            TEST  EAX,EAX
00402054    0F84 E5000000   JE    icl.0040213F
0040205A    E8 D1780000     CALL  icl.00409930
0040205F    0FB6C0          MOVZX EAX,AL
00402062    85C0            TEST  EAX,EAX
00402064    0F84 C4000000   JE    icl.0040212E
```

Оказывается, процедура по адресу 0040204AH выводит строку, содержащую слова "Evaluation copy", а процедура по адресу 0040205AH выводит сообщение о тридцатисуточном ограничении работы с программой. Основываясь на предположении, сделанном в конце предыдущего раздела, обратимся сразу ко второй процедуре (0040205AH). Попытка изменить команду JE на JNE не приводит к желаемому результату. В пошаговом режиме обнаруживаем, что процедура возвращает 1. Делаю простое предположение, что для нас в данной процедуре важно только содержимое регистра EAX. Заменяю CALL icl.00409930 на MOV EAX,1 (количество байтов и в той и в другой команде одинаково) и сохраняю исполняемый модуль на диск. Проверяем, и, о чудо, все работает, в том числе и после истечения тридцати суток.

2.4.4. Пример 4. Снятие защиты

Перед вами пример, как часто до цели добираешься длинным окружным путем вместо того, чтобы быстро пройти по короткой и ровной дороге. На этот раз объектом моего маленького исследования станет программа GetPixel. Программа предназначена для "снятия" с экрана цветовых пикселов. Она попала ко мне уже вместе с программой crack.exe. Для тех, кто не знает: так обычно называют программы для снятия защиты. А поскольку мне необходим был учебный пример, я предпочел снимать защиту без посторонней помощи (и интересно, и полезно). Замечу, что программа написана на языке Visual Basic, но я никак не использую это знание для исследования кода. Во всяком случае известные мне декомпиляторы языка Visual Basic не дают сколько-нибудь полезных результатов.

Стадия 1. Попытка зарегистрироваться

На рис. 2.26 представлено окно программы GetPixel, которое по замыслу автора должно использоваться для регистрации пользователя. Поля **Name** и **Registration Code**[6] предназначены, соответственно, для ввода имени регистрируемого и кода регистрации. При нажатии кнопки **OK** делается проверка имени и пароля. Разумеется, когда я ввел произвольное имя и пароль, программа выдала сообщение, что я ошибся. Иного я и не ожидал.

Ну что же, попробуем заставить программу зарегистрировать мое имя и пароль. По логике предпринимаемых мною действий начнем с поиска строк, содержащих слово Registration.

Открываем знаменитый дизассемблер IDA Pro и загружаем туда нашу программу. В окне **Strings** находим сразу три строки, содержащих данное слово: "Register Successfully!", "Registration", "Register Fail!" Ага, похоже мы на вер-

[6] Не прописанное на рисунке слово Code — не моя вина: так работает программа.

ном пути. Начнем с первой фразы. Щелкнув дважды по строке, оказываемся в нужном месте окна дизассемблера:

```
.text:00409720      aRegisterSucces:  ; DATA XREF: .text:00417ECE?o
.text:00409720      unicode 0, <Register Successfully!>,0
```

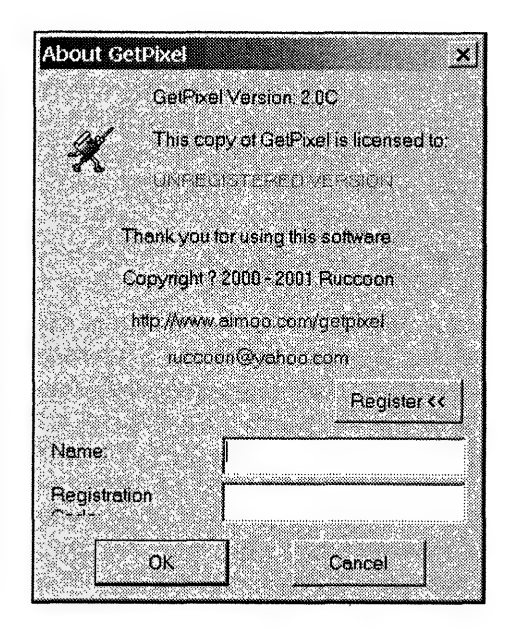


Рис. 2.26. Окно регистрации программы GetPixel

И далее по ссылке находим следующий фрагмент:

```
.text:00417EC5      lea     edx, [ebp-134h]
.text:00417ECB      lea     ecx, [ebp-34h]
.text:00417ECE      mov     dword ptr [ebp-12Ch], offset aRegisterSucces
; "Register Successfully!"
.text:00417ED8      mov     dword ptr [ebp-134h], 8
.text:00417EE2      call    ds:__vbaVarDup
```

Что собой представляет функция `__vbaVarDup`, сказать трудно, но похоже на сообщение — сообщение об удачной регистрации. Попробуем подробнее изучить текст программы вблизи данных строк. Чуть выше по тексту фрагмента обнаруживаем:

```
.text:00417E76      push    ecx
.text:00417E77      push    edx
.text:00417E78      push    4
```

```
.text:00417E7A      call    edi ; __vbaFreeVarList
.text:00417E7C      add     esp, 20h
.text:00417E7F      cmp     [ebp-1A8h], bx
.text:00417E86      jz      loc_4181A4
```

Это уже настораживает. Посмотрим, что находится по адресу `loc_4181A4`. Переходим и чуть ниже обнаруживаем еще один фрагмент:

```
.text:00418268      mov     dword ptr [ebp-12Ch], offset aRegisterFailed
; "Register Failed!"
.text:00418272      mov     dword ptr [ebp-13Ch], offset aPleaseVisit
; "Please visit"
.text:0041827C      mov     dword ptr [ebp-14Ch], offset aHttpWww_aimoo_
; "http://www.aimoo.com/getpixel"
.text:00418286      mov     dword ptr [ebp-15Ch], offset aToGetYourRegis
; "to get your register code"
.text:00418290      call    ebx ; __vbaVarCat
```

Ага, сообщение о неудачной регистрации и приглашение на сайт. Да, похоже мы на верном пути. Запускаем hiew32.exe и находим адрес `.text:00417E86`. Вводим 6 байтов `90h` (`NOP`). Выходим, запускаем программу. Далее в окне регистрации вводим произвольное имя и код и получаем сообщение, что зарегистрированы. Ура, задача решена!? Однако не тут-то было.

Стадия 2. Избавляемся от надоедливого окна

Но трудности этим не заканчиваются. Пока после регистрации я не выходил из программы, окно регистрации сообщало, что мы зарегистрировались. После перезапуска окно опять стало показывать, что копия не зарегистрирована. Кроме этого, при перезапусках с некоторой вероятностью стало появляться окно, представленное на рис. 2.27. При нажатии кнопки **Да** программа пытается выйти на сайт создателя, в случае нажатия кнопки **Нет** программа продолжает свою работу обычным способом.

В том же каталоге, где расположена программа, я обнаруживаю файл cklickme.reg, который содержит скрипт для записи в реестр правильного имени и пароля, если, конечно, мы их знаем. Обращаюсь к реестру по найденному в скрипте адресу. Оказалось, что-то имя и пароль, которые мы ввели, записались именно туда. По-видимому, программа при запуске сравнивает их с некоторыми эталонными значениями, которые мы не знаем и, забегая вперед, не узнаем, а далее указывает в окне, что программа так и не зарегистрирована. Кроме этого, с некоторой вероятностью при запуске выдается окно-"надоедало" (рис. 2.27).

Но давайте действовать по порядку, и разберемся сначала с надоедливым окном. Начнем с поиска строки "How do you feel me?" в окне **Strings** IDA

Pro. Легко находим ее и обращаемся к участку кода, где есть ссылка на эту строку. Вот эти строки:

```
.text:0040B217      push    eax
.text:0040B218      mov     dword ptr [ebp-0D0h], offset aHowDoYouFeelMe
; "How do you feel me?"
.text:0040B222      mov     dword ptr [ebp-0E0h], offset aIWantToBeConfi
; "I want to be confirmed :-)"
.text:0040B22C      call    esi ; __vbaVarCat
.text:0040B22E      lea     ecx, [ebp-0E8h]
.text:0040B234      push    eax
.text:0040B235      lea     edx, [ebp-98h]
.text:0040B23B      push    ecx
.text:0040B23C      push    edx
.text:0040B23D      call    esi ; __vbaVarCat
.text:0040B23F      push    eax
.text:0040B240      call    ds:rtcMsgBox
```

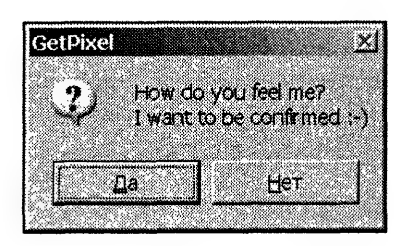


Рис. 2.27. Окно-"надоедало"

Очевидно, что `call rtcMsgBox` — это как раз вызов функции `MessageBox`. Разумеется, эта функция нам не нужна и мы можем забить ее `NOP`. Но погодите. Ведь она должна предлагать выбор, и мы должны выбрать **Нет**. Опустимся немного вниз. Вот этот фрагмент:

```
.text:0040B2B6      call    ds:__vbaVarTstEq
.text:0040B2BC      test    ax, ax
.text:0040B2BF      jz      short loc_40B305
.text:0040B2C1      mov     esi, ds:__vbaStrToAnsi
.text:0040B2C7      push    1
.text:0040B2C9      lea     edx, [ebp-60h]
.text:0040B2CC      push    offset aC        ; "C:\\"
```

```
.text:0040B2D1          push    edx
.text:0040B2D2          call    esi ; __vbaStrToAnsi
.text:0040B2D4          push    eax
.text:0040B2D5          push    0
.text:0040B2D7          lea     eax, [ebp-5Ch]
.text:0040B2DA          push    offset aHttpWww_aimoo_ ;
"http://www.aimoo.com/getpixel"
.text:0040B2DF          push    eax
.text:0040B2E0          call    esi ; __vbaStrToAnsi
.text:0040B2E2          push    eax
.text:0040B2E3          push    0
.text:0040B2E5          push    0
.text:0040B2E7          call    sub_407CB0
.text:0040B2EC          call    ds:__vbaSetSystemError
```

Очевидно, что если условие равенства нулю содержимого регистра EAX не выполнится, то как раз и будет вызываться сайт автора. Таким образом, следует заменить JZ на JMP SHORT — и все.

Запускаем hiew32.exe и вносим изменения в два указанных фрагмента. Не забываем, что забивать надо и сам вызов процедуры, и команды PUSH к ней. В результате действительно раздражающее меня окно перестает запускаться.

Стадия 3. Доводим регистрацию до логического конца

Ну что же, теперь осталось последнее — заставить программу поверить, что в реестре находятся правильные регистрационные данные. По-видимому, разумно предположить, что есть какая-то процедура, которая и проверяет правильность пароля.

Признаюсь, что тут я плутал около часа, используя попеременно то дизассемблер, то отладчик OllyDbg. Мне надо было сразу сообразить, как добраться до этой процедуры. Вот этот путь я вам сейчас и опишу.

Прежде всего, надо было обратить внимание на название полей в реестре, которые заполняются при регистрации. Это поля **License** и **RegUser**. В поле **License** как раз пароль и записывается. Вот и поищем эту строку.

Строку мы обнаруживаем. Она встречается в двух местах. Это уже обнадеживает: к паролю обращаются при запуске программы и из окна регистрации. Глядим на дизассемблированный текст и видим, что в одном случае используется функция rtcGetSetting, а во втором — функция rtcSaveSetting. Ну, тут уже все ясно. Первая функция читает пароль, а

вторая записывает. Данные об этих функциях есть в MSDN. Очевидно, что нам следует обратить внимание именно на первую функцию.

Переходим к нужному фрагменту и, спускаясь по тексту вниз, пытаемся понять логику программы. Двигаясь вниз, будем отслеживать только небиблиотечные процедуры. Какая-то из них, скорее всего, и будет процедурой проверки правильности пароля и имени.

Вначале мое внимание привлек следующий фрагмент:

```
.text:0040AE00        lea      edx, [ebp-78h]
.text:0040AE03        push     edx
.text:0040AE04        call     sub_415160
```

Что помещают в EDX? Запускаем отладчик, ставим точку прерывания на адрес 40ae03. Смотрим, на что указывает в стеке регистр EDX. Оказалось, что на имя, полученное из реестра. Пароля здесь нет. Следовательно, процедура не та, что нам нужна. И мы двигаемся далее. А вот это уже интересно:

```
.text:0040AE5C        lea      edx, [ebp-88h]
.text:0040AE62        lea      eax, [ebp-78h]
.text:0040AE65        push     edx
.text:0040AE66        push     eax
.text:0040AE67        call     sub_416070
```

Из отладчика узнаем, что EDX указывает на строку, состоящую из имени и пароля — где-то по дороге их объединили. Выполняем в отладчике процедуру — она возвращает 0 в регистре EAX, а 0 во многих языках — это false. Ну, что же, пожалуй, пришло время эксперимента.

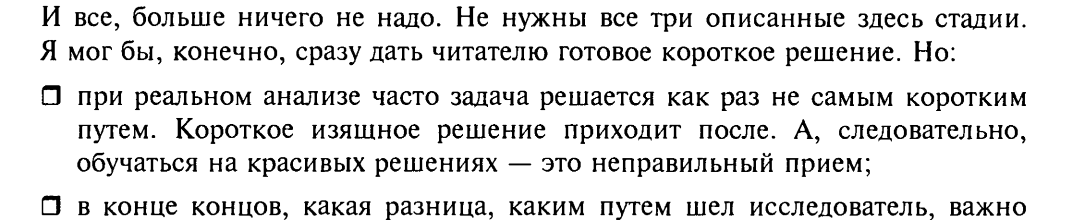
Запускаем hiew32.exe и вместо фрагмента

```
.text:0040AE65        push     edx
.text:0040AE66        push     eax
.text:0040AE67        call     sub_416070
```

ставим команду MOV EAX,1, а остальные байты забиваем байтами 90H. Запускаем программу, входим в окно регистрации... И, о радость, мы зарегистрированы!

Стадия 4. Неожиданная развязка

Теперь я вам скажу, что имеется куда более короткая дорога к правильному результату. Ну, конечно, наверное, вы уже догадались. Нужно просто обратиться по адресу 00416070H, т. е. адресу, где начинается процедура проверки пароля, и в самом начале ее поставить всего две команды: MOV EAX,1, RETN 8.

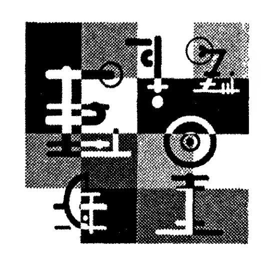


И все, больше ничего не надо. Не нужны все три описанные здесь стадии. Я мог бы, конечно, сразу дать читателю готовое короткое решение. Но:

- при реальном анализе часто задача решается как раз не самым коротким путем. Короткое изящное решение приходит после. А, следовательно, обучаться на красивых решениях — это неправильный прием;
- в конце концов, какая разница, каким путем шел исследователь, важно другое — задача решена.

Я думаю, у читателя возник еще один, более частный вопрос. В своем решении я основывался на информации, полученной из найденного в каталоге скрипта. Из него я узнал, где записываются имя и пароль при регистрации программы. А если бы не было скрипта? Да нет проблем! Можно воспользоваться каким-нибудь монитором, отслеживающим доступ к реестру. А если нет монитора, то и прямой анализ дизассемблируемого текста вполне годится. Функции `rtcSaveSetting` и `rtcGetSetting` так и напрашиваются для анализа.

Глава 3



Основные парадигмы анализа исполняемого кода

В *разд. 2.4* я привел несколько простых примеров анализа и исправления исполняемого кода. Задача данной главы — заложить некоторую теоретическую базу. Опираясь на нее, можно переходить и к более сложным случаям исследования.

Говоря об анализе кода, надо понимать, что это не синоним декомпиляции, т. е. перевода двоичного исполняемого кода в программу на языке высокого уровня. И хотя в данной главе мы будем говорить о программных конструкциях языков высокого уровня, все же нашей главной целью является не восстановление исходного текста программы, что в общем случае нельзя сделать, а понимание логики программы. Примеры, приведенные в *разд. 2.4*, являются демонстрацией анализа кода, направленной на решение вполне конкретных задач (анализ кода в заданном контексте). Мы решили эти задачи, абсолютно не пытаясь понять, какие программные конструкции того или иного языка программирования были использованы. Но при решении более сложных задач нам не обойтись без знания этих конструкций и понимания того, как эти конструкции отражаются в двоичном коде после компиляции.

Даже у одного языка программирования может существовать множество компиляторов. К таковым, в частности, относится язык программирования C++. Кроме этого, у каждого компилятора, как правило, существует несколько режимов компиляции, обычно связанных со способами оптимизации результирующего кода, а также добавления в исполняемый код различных проверочных процедур (например, проверка выхода за границы буфера). Сказанное я представил на рис. 3.1. Изучить всю изображенную на рисунке иерархию в общем случае невозможно, да и не нужно. Надо понять лишь закономерности формирования исполняемого кода.

Надеюсь, что материал, представленный в данной главе, поможет овладеть этими закономерностями. Весь анализ исполняемого кода в данной главе производится на основе лучшего в настоящее время дизассемблера IDA Pro. В *главе 5* можно найти справочные материалы по этой программе.

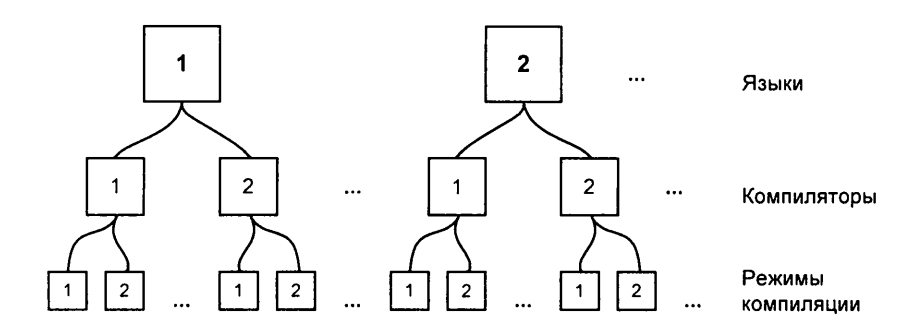


Рис. 3.1. Иерархия "язык — исполняемый код"

3.1. Идентификация данных

Идентификацию данных мы уже обсуждали в *разд. 1.6*, но там я говорил о языке ассемблера. Анализировать код, написанный на языке ассемблера, и проще, и сложнее. Проще, потому что вы пишете тот самый код, который затем и окажется в откомпилированной программе. Труднее, потому что язык ассемблера не накладывает на программиста практически никаких ограничений, и, значит, все зависит от внутренней дисциплины программиста и задач, которые он ставит перед собой. Если вам требуется запутать будущего исследователя вашего кода, то лучше языка ассемблера вы не найдете. Когда мы пишем программу на языке высокого уровня, то, что в результате получится после компилирования вашего текста, абсолютно не ясно. Более того, большинство программистов, пишущих, скажем, на Visual C++ или Delphi, никогда и не задумываются над тем, что из их программы сделает компилятор. При анализе такого кода приходится "продираться" через:

- особенности работы компилятора;
- стиль самого программиста.

Данный раздел посвящен вопросу идентификации данных, используемых в языках высокого уровня.

3.1.1. Глобальные переменные

Есть мнение, что глобальные переменные вредны для программирования. Однако ими пользуются и будут пользоваться, а, следовательно, нам надо учиться их распознавать.

Влияния оптимизации

Об оптимизации по скорости выполнения и объему кода

Начнем наши изыскания с очень простой программы, написанной на языке С++[1]. Она представлена в листинге 3.1. Программа содержит три глобальные переменные. Одна из трех глобальных переменных не инициализированная.

Листинг 3.1

```
#include <stdio.h>
int a, b=20, s=0;
void main()
{
  a=10;
  s=a+b;
  printf("%d",s);
};
```

Посмотрим, что из этого простого текста сделает компилятор Visual С++ (Visual Studio .NET 2003). Загрузим исполняемый модуль, откомпилированный с опцией "оптимизация отсутствует", в дизассемблер IDA Pro. В листинге 3.2 представлен дизассемблированный текст. Надеюсь, вы легко разберетесь в дизассемблированном тексте, который я снабдил также своим кратким комментарием.

Листинг 3.2

```
.text:00401000 _main  proc near             ;CODE XREF: start+16E?p
.text:00401000        push  ebp
.text:00401001        mov   ebp, esp
```

[1] Можно было бы сказать и С, поскольку здесь не используются никакие возможности, появившиеся в языке С++. Не будем обращать на эти нюансы внимание, тем более что все равно будем иметь в виду два наиболее известных компилятора — Visual С++ и Borland С++.

```
.text:00401003     mov   dword_4086E0, 0Ah  ;a=10
.text:0040100D     mov   eax, dword_4086E0  ;a->eax
.text:00401012     add   eax, dword_408040  ;a+b->eax
.text:00401018     mov   dword_4086E4, eax  ;eax->s
.text:0040101D     mov   ecx, dword_4086E4  ;s->ecx
.text:00401023     push  ecx
.text:00401024     push  offset unk_4060FC  ;формантная строка printf
.text:00401029     call  _printf
.text:0040102E     add   esp, 8
.text:00401031     xor   eax, eax
.text:00401033     pop   ebp
.text:00401034     retn
.text:00401034 _main  endp
```

Комментарий к листингу 3.2

Какие интересные моменты можно почерпнуть из листинга 3.2?

- Прежде всего, мы видим, что IDA Pro прекрасно справился с распознаванием глобальных переменных. Впрочем, что же здесь удивительного? В тексте есть прямые ссылки на глобальные переменные (dword_4086E0, dword_4086E4, dword_408040). Ассемблерные же команды вполне определенно говорят нам о размере переменных. Размер переменных — весьма важная проблема дизассемблирования. Далеко не всегда удается четко узнать этот размер. Обратите внимание, что переменная b (dword_408040) расположена отдельно от двух других переменных. Дело в том, что компилятор считает переменные a (dword_4086E0) и s (dword_4086E4)[2] неинициализированными переменными. Но об этом будем подробно говорить в *разд. "Размер, расположение и тип переменных" далее в этой главе.*
- Даже не очень искушенный программист обнаружит, что откомпилированный текст страдает избыточностью:
  - это *пролог* (PUSH EBP/MOV EBP,ESP) и *эпилог* (POP EBP) функции, о которых мы будем говорить в *разд. 3.2.1.* В данной функции эти элементы абсолютно лишние, поскольку регистр EBP используется для адресации стековых переменных и параметров, а таковые в данной программе отсутствуют;
  - переменная a инициализируется, а затем используется при сложении. Поскольку ее значение не выводится и никак более не используется, то вместо переменной a можно использовать простую константу;

[2] Начальная инициализация s не имеет никакого смысла, т. к. начальное значение s нигде не используется.

* бросается также в глаза абсолютно ненужное использование для переменной s области памяти. Действительно, поскольку результат сложения оказывается в регистре EAX, то далее логично именно его использовать в качестве переменной s. Другими словами, переменную s неплохо бы сделать регистровой.

На листинге 3.3 представлен дизассемблированный код той же самой программы (см. листинг 3.1), которая была откомпилирована с опцией "создавать быстрый код". Как видим, исчезли пролог и эпилог функции. Ну, да ладно, о них пока речь не идет. Интересно, как получается сумма (переменная s). Суммирование осуществляется путем сложения содержимого регистра с константой, что, разумеется, выполняется гораздо быстрее, чем сложение регистра с переменной. Кроме того, обращает на себя внимание группировка команд. Вначале значения отправляются в стек, а затем идут две команды пересылки данных. Этот прием, основанный на свойствах процессора Pentium, называется *спариванием*. Суть его заключается в том, что две команды, удовлетворяющие определенным условиям, выполняются параллельно друг другу. Другими словами, две команды выполняются по скорости как одна. Итак, часть наших пожеланий компилятором выполнена.

Замечание

Современные Intel-совместимые процессоры имеют два конвейера для выполнения инструкций. Их называют U-конвейерами и V-конвейерами. При определенных обстоятельствах процессор будет выполнять две идущие друг за другом команды в разных конвейерах, в результате скорость выполнения практически удвоится. Существуют инструкции, которые могут выполняться только в конвейере U, другие инструкции могут выполняться только в конвейере V, наконец, есть инструкции, которые могут выполняться в обоих конвейерах. Зная это, можно группировать команды процессора так, что скорость выполнения программы будет значительно увеличена. Современные компиляторы "знают" эту особенность процессоров, поэтому, встретив в исполняемом коде необычный порядок следования инструкций, в первую очередь следует вспомнить о спаривании инструкций.

Листинг 3.3

```
.text:00401000 _main  proc near                   ; CODE XREF: start+16E?p
.text:00401000        mov   eax, dword_408040   ;b->eax
.text:00401005        add   eax, 0Ah ;сумма здесь
.text:00401008        push  eax
.text:00401009        push  offset unk_4060FC
.text:0040100E        mov   dword_4086E0, 0Ah  ;10->a
.text:00401018        mov   dword_4086E4, eax  ;eax->s
```

```
.text:0040101D         call  _printf
.text:00401022         add   esp, 8
.text:00401025         xor   eax, eax
.text:00401027         retn
.text:00401027 _main   endp
```

Попробуем теперь провести оптимизацию по объему кода. Дизассемблирование показывает, что изменение в коде минимально (по отношению к листингу 3.3): команда `ADD ESP,8` меняется на пару команд `POP ECX/POP ECX`, по байту каждая, команда же `ADD ESP,8` занимает три байта.

Для чего я привел все эти примеры? Конечно, не для того, чтобы изучать способы оптимизации. Для этого необходима отдельная книга. Я хочу подготовить вас морально и немного теоретически к тому, что код, который вы будете анализировать, благодаря оптимизации может оказаться весьма необычным. Впрочем, в дальнейшем о многих способах оптимизации мы еще будем говорить, и неоднократно.

Замечание

Из вышесказанного, в частности, вытекает, что тягаться с компилятором в деле оптимизации (особенно по скорости) исполняемого кода — дело совсем непростое. Данный пример достаточно прост, но представьте себе, что ассемблерный код состоит из сотен команд. Оптимизировать его вручную — совсем нелегкая задача. Так что во многих случаях приходится полагаться на компилятор, особенно, если речь идет о таком продукте, как Visual C++, который всегда славился своими оптимизационными способностями.

Оценка времени выполнения

При оптимизации текста программы важным вопросом является оценка времени выполнения того или иного фрагмента кода. Достаточно просто это можно сделать с использованием двух API-функций. Первая функция — `QueryPerformanceCounter`. Единственным ее аргументом является указатель на структуру `LARGE_INTEGER`. При правильном выполнении функции в структуру будет помещено количество тактов, прошедших с начала запуска программы. Вторая функция `QueryPerformanceFrequency` также своим аргументом содержит указатель на структуру `LARGE_INTEGER`. В эту структуру помещается тактовая частота. Итак, если t_1 и t_2 — это количество тактов на начало и конец исследуемого промежутка программы, f_r — тактовая частота, то количество миллисекунд, которые приходятся на выполнение данного программного фрагмента, выразятся формулой:

$$\frac{(t_2 - t_1) \times 1000}{f_r}.$$

Разумеется, речь идет об оценочных результатах, т. к. говорить о точном значении времени выполнения программного фрагмента в многозадачной среде не приходится.

Указатели на глобальные переменные

Язык C невозможно представить без указателей. Это квинтэссенция данного языка и его судьба. Вместо переменной можно оперировать *указателем* на переменную. Для управления указателями компиляторы используют косвенную адресацию. Впрочем, это утверждение достаточно очевидно. Если `s` — это некоторый указатель на данные, тогда выполняем команду `MOV EDX,s`, и доступ к данным осуществляется через `[EDX]`: например, команда `MOV EAX,[EDX]` осуществляет перенос четырехбайтовой величины из области данных в регистр `EAX`.

В программе из листинга 3.4 одна из глобальных переменных определяется указателем. В листинге 3.5 представлен дизассемблированный листинг исполняемого кода программы.

Листинг 3.4

```
#include <stdio.h>
#include <stdlib.h>
int  a, b=20;
int * s;
void main()
{
  s=(int*)malloc(4);
  a=10;
  *s=a+b;
  printf("%d",*s);
  free(s);
};
```

Листинг 3.5

```
.text:00401000 _main   proc  near    ; CODE XREF: start+16E?p
.text:00401000         push  ebp
.text:00401001         mov   ebp, esp
.text:00401003         push  4                 ;резервируем 4 байта
.text:00401005         call  _malloc
```

```
.text:0040100A        add   esp, 4              ;чистим стек
.text:0040100D        mov   dword_4086C0, eax   ;переменная
                                                ;содержит указатель
.text:00401012        mov   dword_4086C4, 0Ah   ;a=10
.text:0040101C        mov   eax, dword_4086C4   ;a -> eax
.text:00401021        add   eax, dword_408040   ;a+b -> eax
.text:00401027        mov   ecx, dword_4086C0   ;адрес указателя в ECX
.text:0040102D        mov   [ecx], eax          ;сумма по адресу, на
                                                ;которую показывает
                                                ;указатель
.text:0040102F        mov   edx, dword_4086C0   ;указатель -> EDX (???)
.text:00401035        mov   eax, [edx]          ;сумма -> EAX
.text:00401037        push  eax                 ;сумма в стек
.text:00401038        push  offset unk_4060FC   ;форматная строка
.text:0040103D        call  _printf
.text:00401042        add   esp, 8
.text:00401045        mov   ecx, dword_4086C0   ;указатель -> ECX
.text:0040104B        push  ecx
.text:0040104C        call  _free               ;освобождаем указатель
.text:00401051        add   esp, 4
.text:00401054        xor   eax, eax
.text:00401056        pop   ebp
.text:00401057        retn
.text:00401057 _main  endp
```

Комментарий к листингу 3.5

Заметим, что в листинге 3.5 дважды применяется косвенная адресация (через регистры ECX и EDX). Второе использование косвенной адресации (через EDX), конечно, вызывает недоумение, ведь в ECX уже содержится адрес переменной s. Так зачем же еще использовать EDX? Впрочем, вопрос-то риторический. Я ведь компилировал программу, указав компилятору, что оптимизации никакой не требуется. Поэтому компилятор сгенерировал один фрагмент для записи через указатель, а второй фрагмент — для чтения через указатель.

Итак, какой вывод можно сделать из всего сказанного? Ответ очевиден: если при беглом просмотре дизассемблированного текста вы видите использование косвенной адресации, следовательно, в исходном тексте программы скорее всего присутствуют указатели.

Глобальные переменные и константы

Приступим к рассмотрению довольно тонкого вопроса: как отличить адрес глобальной переменной от обычной константы?

Посмотрим внимательно на программу из листинга 3.6. Переменным a, b, c присваиваются некоторые числовые константы, а затем выводятся при помощи стандартной библиотечной функции вывода на консоль. Текст на С совершенно корректен, однозначен и не может вызывать никаких двусмысленных толкований. Но посмотрим, как после трансляции исполняемый код будет трактоваться дизассемблером IDA Pro (листинг 3.7).

Листинг 3.6

```
#include <stdio.h>
int a,b,c;
void main()
{
  a=10;
  b=20;
  c=0x4086d0;
  printf("%d %d %d\n",a,b,c);
};
```

Листинг 3.7

```
.text:00401000 _main   proc near             ; CODE XREF: start+16E?p
.text:00401000         push  ebp
.text:00401001         mov   ebp, esp
.text:00401003         mov   dword_4086C8, 0Ah
.text:0040100D         mov   dword_4086C0, 14h
.text:00401017         mov   dword_4086C4, offset unk_4086D0
.text:00401021         mov   eax, dword_4086C4
.text:00401026         push  eax
.text:00401027         mov   ecx, dword_4086C0
.text:0040102D         push  ecx
.text:0040102E         mov   edx, dword_4086C8
.text:00401034         push  edx
.text:00401035         push  offset aDDD    ; "%d %d %d\n"
.text:0040103A         call  _printf
.text:0040103F         add   esp, 10h
```

```
.text:00401042          xor   eax, eax
.text:00401044          pop   ebp
.text:00401045          retn
.text:00401045 _main    endp
```

Комментарий к листингу 3.7

Итак, внимательно посмотрим на листинг 3.7, который мы получили с помощью программы IDA Pro. Метка dword_4086C8 — это, очевидно, переменная a, метка dword_4086C0 — переменная b, метка dword_4086C4 — переменная c. Но что это? В переменную dword_4086C4 засылается адрес ячейки unk_4086D0. Какая ячейка? Число 0x4086D0 — просто константа! Однако IDA Pro посчитал это число адресом. Вот как! Правда, префикс unk_ означает, что дизассемблер все же сомневается в том, что скрывается за этим адресом. Но если дизассемблер сомневается, то при анализе мы должны принять вполне однозначный вывод. Правда, данный текст весьма прост, и сделать однозначный вывод совсем несложно. Мы не будем сомневаться, как это сделал IDA Pro. Как ни странно на первый взгляд, но в данной ситуации дизассемблер W32Dasm оказывается на "высоте", но не в силу своих выдающихся способностей по распознаванию адресов и констант, а в силу того, что таковые способности у него как раз и отсутствуют, и он всё (или почти всё) считает константами.

А как вы думаете, что произойдет, если переменной c будет присвоено значение 0x4086c0? Думаю, вы уже догадались. Ведь дизассемблер IDA Pro получит дополнительное подтверждение, что это адрес какой-то переменной. Вместо команды mov dword_4086C4,offset unk_4086D0 в листинге появится mov dword_4086C4,offset dword_4086c0. Итак, дизассемблер более не сомневается, что перед нами переменная, но мы-то знаем, что это не так. Более того, мы легко сделаем такой вывод из листинга дизассемблера.

Однако вот какой вопрос меня мучает. Дизассемблер — это программа, и ей необходим строгий критерий, который можно было бы реализовать алгоритмически. В нашем случае, где нет команд, которые бы подтверждали (или опровергали), что перед нами адрес, нельзя найти иного критерия, кроме как диапазон, попадание в который делает константу претендентом на адрес. Каков же диапазон в нашем случае? Да все очень просто. Имеется диапазон адресов, который отведен для данных. Попадание в этот диапазон дизассемблер IDA Pro рассматривает как один из признаков адреса данных. Но есть еще диапазон адресов кода. Например, если в нашем случае взять константу равной 0x401000, то дизассемблер посчитает, что перед нами адрес функции _main. Причем IDA Pro не просто станет "подозревать" константу, но будет уверен, что это в действительности адрес.

Итак, каков же вывод? Вывод один — "de omnibus dubitandum", т. е. "подвергай все сомнению". Разумеется, если с подозрительной константой обращаются как с адресом, например, используют команду LEA, здесь уже можно говорить об адресе с большей уверенностью. А уж если вы увидели, что константа затем используется в косвенной адресации или в качестве параметра функции, который по определению является адресом, то здесь сомнения должны оставить вас.

Размер, расположение и тип переменных

Когда-то, очень давно, во времена MS-DOS, в одном пособии по Паскалю я встретил утверждение, что использование однобайтовых переменных вместо двухбайтовых ускоряет работу программы. Я не поленился и заглянул в ассемблерный код программы. Оказалось все совсем не так. Я написал об этом в своей первой книжке "Ассемблер: учебный курс" ([3]). А как обстоит дело сейчас, в 32-битной системе? Есть ли смысл использовать однобайтовые и двухбайтовые переменные вместо четырехбайтовых? Где располагаются переменные, и как дизассемблер может определить их размер? Все это мы рассмотрим в данном разделе.

Начнем с примера, подобного тому, который я разбирал в *разд. 1.1.3.* Вот этот фрагмент:

```
BYTE  e=0xab;
WORD  c=0x1234;
DWORD b=0x34567890;
```

Если обратиться к памяти, то обнаружим, что все переменные выровнены по границе, кратной 4. Но оказывается, что причиной такого выравнивания является всего лишь порядок объявления переменных. Например, если взять следующий порядок

```
WORD  c=0x1234;
BYTE  e=0xab;
DWORD b=0x34567890;
```

то компилятор расположит переменные в памяти так, что первые две переменные окажутся в двух соседних словах. Переменная же b, как и прежде, будет расположена на 4-байтовой границе. Существуют оптимальные требования к выравниванию данных. В табл. 3.1 представлена информация о выравнивании данных различных размеров.

Таблица 3.1. Оптимальные требования к выравниванию данных

Размер данных	Кратность выравнивания	Размер данных	Кратность выравнивания
1 байт	1 (выравнивания нет)	8 байтов	8
2 байта	2	10 байтов	16
4 бата	4	16 байтов	16
6 байтов	8		

Рассмотрим следующий пример (листинг 3.8).

Листинг 3.8

```
#include <stdio.h>
#include <windows.h>
WORD b=10;
BYTE a;
DWORD c;
void main()
{
  a=10;
  c=30;
  printf("%d %d %d\n",a,b,c);
};
```

Пример куда уж проще! Однако здесь есть одна изюминка, которая поможет нам раскрыть некоторые закономерности расположения переменных в памяти. Суть в том, что переменные `a` и `c` — это неинициализированные переменные. Значения им присваиваются прямо в тексте программы. Переменная `b` — инициализированная. Есть ли разница между этими переменными? Оказывается, есть. Откомпилируем программу с помощью компилятора Visual C++ и дизассемблируем исполняемый модуль при помощи IDA Pro. Не утруждая читателя листингами, скажу, что в IDA Pro все переменные расположатся в секции .data. Однако вспомним материал *разд. 1.5.3*, где говорится, что инициализированные переменные помещаются в секцию .data, а неинициализированные переменные — в секцию .bss. Интересно, что из листинга IDA Pro видно, что хоть все переменные находятся в одной секции, но в разных ее частях: в начале располагается инициализированная переменная, а затем, через достаточно большой промежуток, две неинициализированные переменные. Чтобы понять причину такого явления, откомпиляции

пилируем программу с ключом /Fas, что приведет в процессе компиляции к генерированию промежуточного ассемблерного листинга. Просматривая ассемблерный листинг, мы обнаруживаем интересный факт. Действительно в листинге присутствуют два сегмента с именами _data (с инициализированной переменной) и _bss (с неинициализированными переменными), которые впоследствии должны перейти в соответствующие секции. Но дело в том, что компилятор знает хорошо названия сегментов _bss и _data и впоследствии объединяет их в одну секцию .data. При этом данные, расположенные в сегменте _bss, всегда идут после данных сегмента _data. Чтобы проверить это утверждение, напишите простую программу на ассемблере с двумя сегментами данных (_bss и _data). В результате компоновки останется одна секция данных .data. Если же слегка изменить имена сегментов, например, взять _bss1 вместо _bss, то в исполняемом модуле будут две секции: .data и _bss1 (с подчеркиванием).

Мы проверили компилятор Visual C++. А как у других компиляторов? Проверка программы из листинга 3.8 компилятором Borland C++ 5.00 показала полностью аналогичную ситуацию. Ассемблерный код содержал два сегмента данных: _data и _bss.

Обратимся теперь к вопросу: как при дизассемблировании определить размер переменной? Можно дать общий ответ на этот вопрос: это можно сделать на основе анализа команд, оперирующих с этой переменной. И это очевидно, ведь переменная проявляет себя в том, какие действия над ней производятся. Пора нам вспомнить материал *разд. 1.4*, где мы обсуждали формат команд микропроцессора Intel. Рассмотрим простую операцию присвоения числовой переменной некоторого целого значения. На С эта операция выглядит просто так: b=10. Соответственно ассемблерная команда в общем случае будет иметь вид MOV [mem],10. Однако мы знаем, что в ассемблере при таких операциях обязательно требуется указать тип переменной (например, byte ptr). Разумеется, тому есть веская причина. Действительно, есть существенная разница между помещением числа 10 в переменную типа WORD и в переменную типа DWORD. А раз есть отличие на уровне математики, то это как-то должно отражаться в формате команды. Так оно и есть. Вот полные коды команд присвоения для переменных трех типов: BYTE, WORD, DWORD.

```
C605 C8864000       14         MOV byte  ptr [04086C8],20
66   C705 C8864000  0A00       MOV word  ptr [04086C8],10
C705 C4864000       1E000000   MOV dword ptr [04086C4],30
```

Смотрите, все байты MOD R/M у всех трех команд одинаковы. Это и понятно: во всех командах первый операнд — это смещение. Интересно, что код команды, тип операнда которого — WORD, отличается от команды с типом DWORD наличием префикса 66H. Этот префикс и указывает, что операнд име-

ет тип WORD, а не DWORD. Для команды же, где первый операнд имеет тип BYTE, имеется свой код. Итак, совершенно очевидно, как дизассемблер узнает о размере переменной, — он просто анализирует программный код.

Говоря о числовых переменных, мы совсем упустили из виду числа с плавающей точкой. Уделим внимание и им (листинг 3.9).

Листинг 3.9

```
#include <stdio.h>
#include <windows.h>
double s,d;
int i;
void main()
{
  s=0.00;
  d=1.034;
  for(i=0; i<100; i++)
    s=s+i/d;
  printf("%f\n",s);
};
```

Как видим, две переменные имеют тип double. Вспомним материал *разд. 1.1.3*, точнее ту его часть, где мы говорили о вещественных числах. Так вот, формат числа double, который использован в C++, в точности соответствует формату длинного вещественного числа, который поддерживается процессором Intel. Точнее той его половиной, которая традиционно называется *арифметическим сопроцессором (см. разд. 1.2.3)*.

В листинге 3.10 представлен дизассемблированный текст функции main программы из листинга 3.9.

Листинг 3.10

```
.text:00401000 _main        proc near     ; CODE XREF: start+16E?p
.text:00401000              var_8 = qword ptr -8
.text:00401000
.text:00401000              push    ebp
.text:00401001              mov     ebp, esp
.text:00401003              fld     ds:dbl_408108
.text:00401009              fstp    dbl_40A9D0
.text:0040100F              fld     ds:dbl_408100
```

```
.text:00401015              fstp    dbl_40A9C0
.text:0040101B              mov     dword_40A9C8, 0
.text:00401025              jmp     short loc_401034
.text:00401027 loc_401027:                  ; CODE XREF: _main+55?j
.text:00401027              mov     eax, dword_40A9C8
.text:0040102C              add     eax, 1
.text:0040102F              mov     dword_40A9C8, eax
.text:00401034
.text:00401034 loc_401034:                  ; CODE XREF: _main+25?j
.text:00401034              cmp     dword_40A9C8, 64h
.text:0040103B              jge     short loc_401057
.text:0040103D              fild    dword_40A9C8
.text:00401043              fdiv    dbl_40A9C0
.text:00401049              fadd    dbl_40A9D0
.text:0040104F              fstp    dbl_40A9D0
.text:00401055              jmp     short loc_401027
.text:00401057 ; ----------------------------------------
.text:00401057
.text:00401057 loc_401057:                  ; CODE XREF: _main+3B?j
.text:00401057              fld     dbl_40A9D0
.text:0040105D              sub     esp, 8
.text:00401060              fstp    [esp+8+var_8]
.text:00401063              push    offset unk_4080FC
.text:00401068              call    _printf
.text:0040106D              add     esp, 0Ch
.text:00401070              xor     eax, eax
.text:00401072              pop     ebp
.text:00401073              retn
.text:00401073 _main        endp
```

Комментарий к листингу 3.10

Прокомментируем текст, который был создан дизассемблером IDA Pro.

- Пропускаем странную переменную `var_8`. О ней будет сказано далее. Пропускаем также пролог функции. Четыре следующие команды весьма примечательны. Это есть не что иное, как присвоение переменным `s` и `d` начальных значений. Для этого компилятор заранее зарезервировал место для двух вещественных констант: `dbl_408108` и `dbl_408100`. С помощью последовательности двух команд `fld` и `fstp` (команды можно найти в табл. 1.19) константа загружается в соответствующую переменную. И кон-

станты, и переменные (dbl_40A9D0 и dbl_40A9C0) занимают, естественно, по восемь байтов. Следующая команда, обнуляющая целую переменную dword_40A9C8, вполне очевидна — это просто присвоение начального значения параметру цикла.

- Далее следует переход в тело цикла на метку loc_401034. Перед меткой идут три команды, назначение которых заключается в увеличении параметра цикла (i++). Вот почему первый раз мы их пропускаем. Проверка на возможное окончание цикла осуществляется командами cmp dword_40A9C8, 64h/jge short loc_401057. Разумеется, 64h — это просто 100.
- Затем идут четыре команды, назначение которых угадывается по тексту исходной программы. Это просто s=s+i/d. Давайте разберем алгоритм. Команда fild dword_40A9C8 загружает целый параметр цикла в вершину стека сопроцессора ST(0). Следующая команда fdiv осуществляет деление параметра на переменную dbl_40A9C0 (это d, разумеется). Далее команда fadd осуществляет сложение результата деления с переменной dbl_40A9D0, где и будет накапливаться сумма. Наконец, поскольку результат сложения находится в стеке сопроцессора, то его командой fstp помещают в переменную dbl_40A9D0. При этом происходит выталкивание стека сопроцессора, т. е. например, содержимое ST(1) переходит в ST(0). Далее безусловный переход возвращает нас на начало цикла.
- Затем идет вызов функции printf. И вот самое интересное. Ведь в стек следует отправить вещественное число. Весьма поучительный прием. Итак, команда fld dbl_40A9D0 посылает вычисленную сумму в стек сопроцессора. Следующая команда — sub esp,8 — резервирует место в стеке для 8-байтовой величины. Команда равносильна двум командам PUSH. А далее команда fstp [esp+8+var_8] помещает сумму из стека сопроцессора в обычный стек. Наконец, следующая команда — push — посылает в стек форматную строку.

Разобранный нами случай, когда начальные значения вещественных переменных хранятся в константах, а затем загружаются в переменную, практикуется компилятором Visual C++. Компилятор Borland C++ использует другой, менее наглядный прием. Вот что можно увидеть в дизассемблированном коде, созданном компилятором Borland C++:

```
.text:0040111B    mov  dword ptr dbl_40C2C4, 95810625h
.text:00401127    mov  dword ptr dbl_40C2C4+4, 3FF08B43h
```

Как видим, в память загружаются две странные константы. И попробуй пойми, что это вещественное число, а потом определи это число. Тут без знаний *разд. 1.1.3* никак не обойтись. Однако и в Visual C++ вы встретитесь с такой же проблемой, если будете оперировать переменными типа float.

Это короткое вещественное число и занимает всего 32 бита. По этой причине для присвоения переменной такого типа и получения ее значения используется обычная команда `MOV`. Но, разумеется, для выполнения каких-либо действий над ними служат команды арифметического сопроцессора. В общем, настоятельно рекомендую разобраться в структуре вещественных чисел *(см. разд. 1.1.3)*.

Итак, увидев команды арифметического сопроцессора, знайте, придется повозиться с вещественными переменными.

Когда мы имеем дело с целыми переменными, то важным вопросом является их знаковость. Как, например, отличить тип `int` от типа `unsigned int` (`DWORD`)? Общий ответ гласит, что следует проанализировать действия, которые осуществляются над переменными, и отсюда вывести их тип. Более конкретным способом определения типа целых переменных является анализ условных конструкций, в которых они участвуют. Например, команда условного перехода `JL` применяется при сравнении чисел со знаком, а ее беззнаковым аналогом является команда `JB`.

Остается ответить еще на один вопрос. Дает ли какой-то выигрыш использование целых переменных меньшего размера, чем четыре байта? Ответ на данный вопрос таков:

- несомненно, использование переменных меньшего размера экономит память;
- однако в конечном итоге это приводит к усложнению алгоритма в откомпилированном коде, т. к. в программе все равно приходится использовать 32-битные переменные. Усложнение же алгоритма в конечном итоге ведет к проигрышу в скорости выполнения и увеличению требуемой памяти.

Сложные типы данных

Строки

Под *строковым типом* в языках программирования понималась некоторая закодированная последовательность символов. Обычно используется ASCII-кодирование. В нем на символ отводится всего один байт. Сейчас все чаще применяется кодировка Unicode, в которой на символ отводятся два байта.

Строка похожа на массив. Различие между ними заключается в том, что структура строки содержит информацию, с помощью которой можно легко определить ее длину. Существуют всего два подхода.

- Конец строки должен быть как-то обозначен. Для этой цели может быть использован какой-то конкретный код — один или несколько байтов. В языке С традиционно для этой цели применяется код 0 (не путать с

символом 0). При использовании кодировки Unicode в конце ставятся соответственно два нуля (символ с нулевым кодом). Кроме этого, некоторые современные компиляторы, приспосабливая строки для обработки двойными словами, могут заканчивать строку целой последовательностью из семи нулевых байтов. Учитывая возросшие в последнее время ресурсы оперативной памяти, такой подход кажется совсем нерасточительным. Данный механизм обладает двумя недостатками:

- для того чтобы узнать длину строки, необходимо просмотреть ее всю, сколь бы длинной она ни была. Да и все строковые операции должны основываться на проверке конца строки, что, конечно, несколько замедляет эти операции;
- данный подход не позволяет использовать непосредственно в строке нулевые байты.

- Где-то должна храниться информация о длине (или конце) строки. Естественно для этой цели использовать байты в начале строки. Так поступают в Паскале и, соответственно, в Delphi. Это может быть всего один байт, и тогда длина строки может составить не более 255 символов. В Delphi, однако, допустимо создавать строки с четырехбайтовым указателем длины. В этом случае возможная длина строки сравнима с объемом адресного пространства, предоставляемого процессу в операционной системе Windows.

Кроме двух описанных подходов возможна и смешанная система, когда перед строкой хранится ее длина, но одновременно в конце строки содержится ограничитель. Такой подход хорош для совместимости, но плох как всякий подход с избыточной информацией, доставляющей головную боль программистам[3].

Замечание

Программисты, работавшие в MS-DOS, несомненно, помнят и функцию с номером 9 21-го прерывания (`int 21h`), с помощью которой можно было вывести на текстовый экран строку символов. В качестве конца строки эта системная процедура воспринимала знак доллара `$`. Такой разделитель, разумеется, не удобен и уже давным-давно не используется.

Начнем с простого примера (листинг 3.11) использования строки в кодировке Unicode.

---

[3] Как, например, быть, если информация о длине строки не соответствует тому, где расположен конец строки?

Листинг 3.11

```
#include <stdio.h>
wchar_t  s[]=L"Hello, programmer!";
wchar_t  f[]=L"%s\n";
void main()
{
    wprintf(f,s);
};
```

Напоминаю, что `wchar_t` задает тип строки в кодировке Unicode; `L` — макрос, преобразующий строку в кодировке ASCII в строку в кодировке Unicode; `wprintf` — функция для вывода строки в кодировке Unicode на консоль, аналог `printf`. Замечу, что строку формата (`f`) для функции `wprintf` также следует представить в кодировке Unicode. Вот как дизассемблирует IDA Pro вызов функции `wprintf`:

```
.text:00401003    push  offset aHelloProgramme ; "Hello, programmer!"
.text:00401008    push  offset aS              ; "%s\n"
.text:0040100D    call  _wprintf
```

Не правда ли, здорово! IDA Pro прекрасно распознал строку в кодировке Unicode. Вот эти строки в секции данных:

```
.data:00409040    aHelloProgramme:          ; DATA XREF: _main+3?o
.data:00409040      unicode 0, <Hello, programmer!>,0
```

При желании, нажав клавишу <A>, можно перевести эту строку в последовательность символов ASCII и обнаружить, что коды символов, находящиеся в промежутке от 0 до 127, переходят в Unicode без изменения путем дополнения байта до слова (добавление старшего нулевого байта). Так что преобразование англоязычного текста из кодировки ASCII в кодировку Unicode — дело совсем простое.

Следующий наш пример касается Delphi[4] (листинг 3.12).

Листинг 3.12

```
var
  s1:widestring;
  s2:string; {по умолчанию это AnsiString}
  s3:shortstring;
```

[4] Здесь и далее я использую компилятор Delphi из пакета Borland Delphi 7.0.

```
begin
  s1:='Hello, world!';
  s2:='Hello, programmers!';
  s3:='Hello, hackers!';
  writeln(s1);
  writeln(s2);
  writeln(s3);
end.
```

Итак, в нашей программе использованы три вида строк, которые представлены в Delphi. Что-то будет, когда мы запустим IDA Pro? Что может быть интереснее программирования?! Только исследование исполняемого кода.

Приступим. Программа загружена и автоматический анализ произведен. Попытаемся найти наши строки в окне **Strings**. Интересные дела, в окне оказалось только строка "Hello, world!". Впрочем, есть надежда, что остальные строки где-то поблизости, и мы их быстро найдем. Так и есть, вот интересующий нас фрагмент:

```
CODE:0044CC4D             align 10h
CODE:0044CC50             dd 18h
CODE:0044CC54 aHelloWorld:
CODE:0044CC54                         ; DATA XREF: sub_44CBAC+21?o
CODE:0044CC54             unicode 0, <Hello world!>,0
CODE:0044CC6E             align 10h
CODE:0044CC70             dd 0FFFFFFFFh, 12h
CODE:0044CC78 aHelloProgarmme db 'Hello, programmers!',0
CODE:0044CC78                         ; DATA XREF: sub_44CBAC+30?o
CODE:0044CC8B             align 4
CODE:0044CC8C dword_44CC8C  dd 6C65480Eh, 68206F6Ch, 656B6361h, 217372h
CODE:0044CC8C                         ; DATA XREF: sub_44CBAC+3A?o
```

Ага! Строку `s2` дизассемблер также распознал, а вот почему он не поместил ее в окно **Strings** — это на его совести. Интересно, что располагается по адресу `0044CC8C` — ссылка на этот блок из текста программы также имеется. Установим курсор на эту строку и нажмем клавишу <A> (можно также обратиться к пункту меню **Options | Ascii string style** и в появившемся окне нажать кнопку **Pascal style**). И, о чудо:

```
CODE:0044CC8C aHelloHackers  db 14,'Hello, hackers!'
CODE:0044CC8C                        ; DATA XREF: sub_44CBAC+3A?o
CODE:0044CC9B                db 0
```

Строка, как видим, обнаружена. Почему она не была найдена дизассемблером сразу? По-видимому, все дело в байте 14, на который была ссылка. Это, несомненно, длина. Но дизассемблер, анализируя ссылку, посчитал, что если это начало строки, то в тексте не может быть символа с кодом 14. В принципе, посчитал-то он правильно, только вот додуматься до того, что это длина строки, он не смог.

Итак, можно делать первые выводы: в случае короткой строки (`shortstring`) ссылка осуществляется на байт длины. Кстати, обратите внимание, что в конце строки стоит код 0; в длину строки, как и должно быть, он не входит.

Продолжим рассматривать две другие строки. Строка по адресу `0044CC78` также имеет 0 на конце. Ссылка, заметим, делается на начало строки, и в конце опять код 0. А где же длина строки? Вот тут интересно. Перед строкой две четырехбайтовых величины. Число `12h`, несомненно, — длина строки. На длину, как мы видим, отводится 4 байта. Но оказывается, в структуру строки входят еще 4 байта. Это так называемый *счетчик ссылок* (reference count). Запомним, для строки этого типа ссылка указывает непосредственно на содержимое строки. Перед самой же текстовой информацией имеется еще 8 байтов дополнительной информации.

Последний тип строки — это строка в кодировке Unicode. Она начинается по адресу `0044CC54`. В отличие от предыдущего случая, в структуру строки входит четырехбайтовая длина, но нет счетчика ссылок. И опять ссылка из текста программы указывает именно на содержимое строки, очевидно, по этой причине дизассемблер и обнаружил данную строку. В конце строки стоят два нулевых байта.

Завершая рассмотрение строк, разберем следующую простую программу (листинг 3.13). Откомпилируем эту программу с помощью Visual C++.

Листинг 3.13

```
#include <stdio.h>
#include <string.h>
char s[]="Good-bye!";
void main()
{
  strcat(s," My love!");
  printf("%s\n",s);
}
```

В листинге 3.14 мы видим дизассемблированный текст программы из листинга 3.13.

Листинг 3.14

```
.text:00401000 _main         proc near     ; CODE XREF: start+16E?p
.text:00401000               push    ebp
.text:00401001               mov     ebp, esp
.text:00401003               push    offset aMyLove  ; char *
.text:00401008               push    offset aGoodBye ; char *
.text:0040100D               call    _strcat
.text:00401012               add     esp, 8
.text:00401015               push    offset aGoodBye ; "Good-bye!"
.text:0040101A               push    offset aS       ; "%s\n"
.text:0040101F               call    _printf
.text:00401024               add     esp, 8
.text:00401027               xor     eax, eax
.text:00401029               pop     ebp
.text:0040102A               retn
.text:0040102A _main         endp
```

Листинг 3.14 достаточно прост, чтобы его комментировать подробно. Заметим только, что, как видно из него, обе строки хорошо распознаются IDA Pro.

А теперь в программе из листинга 3.13 сделаем маленькую корректировку. Сделаем переменную `s` локальной (перенесем определение в функцию `main`). После компиляции и дизассемблирования получим следующий, весьма необычный текст (листинг 3.15).

Листинг 3.15

```
.text:00401000 _main  proc near  ; CODE XREF: start+16E?p
.text:00401000    var_C  = byte ptr -0Ch
.text:00401000    var_8  = dword ptr -8
.text:00401000    var_4  = word ptr -4
.text:00401000
.text:00401000    push   ebp
.text:00401001    mov    ebp, esp
.text:00401003    sub    esp, 0Ch
.text:00401006    mov    eax, ds:dword_4060FC
.text:0040100B    mov    dword ptr [ebp+var_C], eax
.text:0040100E    mov    ecx, ds:dword_406100
.text:00401014    mov    [ebp+var_8], ecx
```

```
.text:00401017    mov    dx, ds:word_406104
.text:0040101E    mov    [ebp+var_4], dx
.text:00401022    push   offset aMyLove  ; char *
.text:00401027    lea    eax, [ebp+var_C]
.text:0040102A    push   eax             ; char *
.text:0040102B    call   _strcat
.text:00401030    add    esp, 8
.text:00401033    lea    ecx, [ebp+var_C]
.text:00401036    push   ecx
.text:00401037    push   offset aS       ; "%s\n"
.text:0040103C    call   _printf
.text:00401041    add    esp, 8
.text:00401044    xor    eax, eax
.text:00401046    mov    esp, ebp
.text:00401048    pop    ebp
.text:00401049    retn
.text:00401049   _main   endp
```

Смотрим на листинг 3.15. Действительно код необычный. Дизассемблер определил только одну строку (константу). Однако первый параметр функции `strcat` — это ведь и есть адрес не найденной дизассемблером строки. Тут ничего не попишешь, функция библиотечная и нам хорошо известна. А вот команды с адреса `00401006` по адрес `0040101E` что значат? Очевидно, что они передают в стековую область (наша строка и должна храниться в стеке) десять байтов. Но это как раз длина нашей строки с учетом нулевого байта. Ага, это компилятор так хитро передает строку из секции данных в область стека! Обратимся к памяти по адресу `004060FC`, откуда начинается блок, передаваемый в стек. Вот этот блок:

```
.rdata:004060FC   dword_4060FC  dd 646F6F47h   ; DATA XREF: _main+6?r
.rdata:00406100   dword_406100  dd 6579622Dh   ; DATA XREF: _main+E?r
.rdata:00406104   word_406104   dw 21h         ; DATA XREF: _main+17?r
```

Нажмем клавишу <A> и преобразуем блок в строку в ASCII-формате, и потерянная строка нашлась. Каков же вывод? Дизассемблер не смог определить одну из строк по той причине, что компилятор обращался со строкой просто как с блоком данных.

Массивы

Как мы видели из предыдущего раздела, что хотя строки и имеют структуру, позволяющую определить размер данных, даже такой мощный дизассемблер, как IDA Pro, не всегда способен распознать строку. Что уж тогда гово-

рить о массивах? Ведь размер массива в структуре никак не прописывается. Разумеется, с размером массивов есть проблемы, однако сам массив идентифицируется достаточно четко. Рассмотрим пример. В листинге 3.16 целочисленный массив заполняется целыми числами от 0 до 9. После трансляции в Visual Studio и загрузки исполняемого кода в IDA Pro имеем листинг 3.17.

Листинг 3.16

```
#include <stdio.h>
int a[10];
void main()
{
  for(int i=0;i<10; i++)a[i]=i;
};
```

Листинг 3.17

```
.text:00401000 _main        proc near           ; CODE XREF: start+16E?p
.text:00401000         var_4  = dword ptr -4
.text:00401000              push    ebp
.text:00401001              mov     ebp, esp
.text:00401003              push    ecx
.text:00401004              mov     [ebp+var_4], 0
.text:0040100B              jmp     short loc_401016
.text:0040100D loc_40100D:                      ; CODE XREF: _main+29?j
.text:0040100D              mov     eax, [ebp+var_4]
.text:00401010              add     eax, 1
.text:00401013              mov     [ebp+var_4], eax
.text:00401016 loc_401016:                      ; CODE XREF: _main+B?j
.text:00401016              cmp     [ebp+var_4], 0Ah
.text:0040101A              jge     short loc_40102B
.text:0040101C              mov     ecx, [ebp+var_4]
.text:0040101F              mov     edx, [ebp+var_4]
.text:00401022              mov     dword_4072C0[ecx*4], edx
.text:00401029              jmp     short loc_40100D
.text:0040102B loc_40102B:                      ; CODE XREF: _main+1A?j
.text:0040102B              xor     eax, eax
.text:0040102D              mov     esp, ebp
```

```
.text:0040102F              pop     ebp
.text:00401030              retn
.text:00401030 _main        endp
```

Комментарий к листингу 3.17

- Со способом организации цикла мы уже сталкивались в листинге 3.10. `var_4`, как вы, несомненно, поняли, есть не что иное, как стековая переменная — параметр цикла. Обратите внимание на команду `mov dword_4072C0[ecx*4],edx` — вот это главное. Нет никакого сомнения, что перед нами массив. `dword_4072C0` — начало массива, `ecx` содержит текущее значение индекса, а масштабный коэффициент 4 говорит о том, что размер элементов массива составляет 4 байта. Конечно, в данной программе размер массива очевиден. Но полагаться, что количество элементов массива всегда определяется количеством итераций в цикле, который обрабатывает массив, не стоит. В одном месте программы программист может использовать часть массива, в другом месте — другую часть. Причем эти куски не обязаны иметь то же начало, что и сам массив. С большей вероятностью можно говорить о том, что размер массива не меньше заданной величины.
- Некоторые сложности могут возникнуть при использовании массива в функциях. В функцию ведь передается просто указатель. При этом данный указатель может передаваться и дальше по цепочке функций. И вот в последней функции мы видим, что некий параметр используется как указатель на массив. Для того чтобы теперь найти, где располагается этот массив, придется пройти по цепочке функций в обратную сторону, что, конечно, потребует времени и усидчивости. Впрочем, в таких случаях лучше воспользоваться отладчиком, поставить точку останова на функцию, где указатель ведет себя как указатель на массив, и получить его (указателя) значение. Далее следует снова обратиться к дизассемблеру, найти этот массив по выданному в отладчике адресу и определить ссылки на него из кода программы. После этого можно продолжить анализ исполняемого кода.

Структуры

Структура — это обобщение понятия "массив". Если массив состоит из однородных элементов, то структура может состоять из элементов различных типов. Как и в случае с массивом доступ к элементам структуры осуществляется на основе базового адреса, который определяет начало экземпляра структуры. Но здесь проблема гораздо серьезнее. Понять, что разнородные куски данных в действительности принадлежат одной структуре, бывает очень непросто. Рассмотрим программу, представленную в листинге 3.18.

Листинг 3.18

```
#include <stdio.h>
#include <windows.h>
struct a {
          char s[10];
          BYTE b;
          int i;
};
a a1;
void main()
{
  for(int j=0; j<10; j++) a1.s[j]='A';
  a1.b=10;
  a1.i=10000;
};
```

Откомпилируем программу и перейдем к дизассемблированному IDA Pro тексту (листинг 3.19).

Листинг 3.19

```
.text:00401000 _main    proc near    ; CODE XREF: start+16E?p
.text:00401000    var_4  = dword ptr -4
.text:00401000      push    ebp
.text:00401001      mov     ebp, esp
.text:00401003      push    ecx
.text:00401004      mov     [ebp+var_4], 0
.text:0040100B      jmp     short loc_401016
.text:0040100D loc_40100D:         ; CODE XREF: _main+26?j
.text:0040100D      mov     eax, [ebp+var_4]
.text:00401010      add     eax, 1
.text:00401013      mov     [ebp+var_4], eax
.text:00401016 loc_401016:         ; CODE XREF: _main+B?j
.text:00401016      cmp     [ebp+var_4], 0Ah
.text:0040101A      jge     short loc_401028
.text:0040101C      mov     ecx, [ebp+var_4]
.text:0040101F      mov     byte_4072C0[ecx], 41h
.text:00401026      jmp     short loc_40100D
```

```
.text:00401028 loc_401028:          ; CODE XREF: _main+1A?j
.text:00401028      mov     byte_4072CA, 0Ah
.text:0040102F      mov     dword_4072CC, 2710h
.text:00401039      xor     eax, eax
.text:0040103B      mov     esp, ebp
.text:0040103D      pop     ebp
.text:0040103E      retn
.text:0040103E _main     endp
```

Посмотрите внимательно на текст из листинга 3.19. В тексте мы видим три различных типа данных, которые определяются указателями `byte_4072C0` (массив), `byte_4072CA` (байт), `dword_4072CC` (двойное слово). Ниоткуда не следует, что все эти переменные должны быть объединены в одну структуру. Впрочем, в данном контексте это совершенно неважно. Отсюда следует, что в программе должны быть такие действия, которые выдавали бы структуру, как единое целое.

Рассмотрим программу из листинга 3.20. Как видно, параметром процедуры `init` является как раз структура (структура `a`). Посмотрим, как эта ситуация отразится в исполняемом коде программы (листинг 3.21). Конечно, программа весьма надуманна. Ведь переданная в функцию структура никак не используется, да и обратно не передается.

Листинг 3.20

```
#include <stdio.h>
#include <windows.h>
struct a {
  char s[10];
  BYTE b;
  int i;
};
a a1;
void init(a);
void main()
{
  init(a1);
};
void init(a c)
{
  for(int j=0; j<10; j++) c.s[j]='A';
```

```
  c.b=10;
  c.i=10000;
};
```

В листинге 3.21 представлена главная функция `main` программы. Процедура `sub_401040`, вызов которой осуществляется по адресу `0040102B`, и является нашей функцией `init`. А вот строки перед процедурой довольно интересны. Обратите внимание на команду `sub esp,10h`. Она эквивалентна четырем командам `PUSH`. Но посмотрите, размер нашей структуры как раз составляет 16 байтов. После команды выделения области в стеке идет команда `move eax,esp`. Таким образом, регистр `EAX` указывает на начало области стека. Ну, а далее эта область стека заполняется данными. Впечатление такое, что мы имеем дело просто с четырьмя двойными словами. Да и IDA Pro так считает. Конечно, то, что сразу выделяется 16 байтов (правда длина структуры 15 байтов, но с учетом того, что поле `i` выравнивается по 4-байтной границе, получается 16), несколько настораживает, но абсолютно ничего не доказывает. Чтобы понять, что все-таки отправлено в функцию, необходимо анализировать код этой функции (листинг 3.22).

Листинг 3.21

```
.text:00401000  _main    proc near    ; CODE XREF: start+16E?p
.text:00401000      push    ebp
.text:00401001      mov     ebp, esp
.text:00401003      sub     esp, 10h
.text:00401006      mov     eax, esp
.text:00401008      mov     ecx, dword_4072C0
.text:0040100E      mov     [eax], ecx
.text:00401010      mov     edx, dword_4072C4
.text:00401016      mov     [eax+4], edx
.text:00401019      mov     ecx, dword_4072C8
.text:0040101F      mov     [eax+8], ecx
.text:00401022      mov     edx, dword_4072CC
.text:00401028      mov     [eax+0Ch], edx
.text:0040102B      call    sub_401040
.text:00401030      add     esp, 10h
.text:00401033      xor     eax, eax
.text:00401035      pop     ebp
.text:00401036      retn
.text:00401036 _main        endp
```

Листинг 3.22

```
.text:00401040 sub_401040  proc near        ; CODE XREF: _main+2B?p
.text:00401040    var_4  = dword ptr -4
.text:00401040    arg_0  = byte ptr  8
.text:00401040    arg_A  = byte ptr  12h
.text:00401040    arg_C  = dword ptr  14h
.text:00401040      push    ebp
.text:00401041      mov     ebp, esp
.text:00401043      push    ecx
.text:00401044      mov     [ebp+var_4], 0
.text:0040104B      jmp     short loc_401056
.text:0040104D loc_40104D:                  ; CODE XREF: sub_401040+24?j
.text:0040104D      mov     eax, [ebp+var_4]
.text:00401050      add     eax, 1
.text:00401053      mov     [ebp+var_4], eax
.text:00401056 loc_401056:                  ; CODE XREF: sub_401040+B?j
.text:00401056      cmp     [ebp+var_4], 0Ah
.text:0040105A      jge     short loc_401066
.text:0040105C      mov     ecx, [ebp+var_4]
.text:0040105F      mov     [ebp+ecx+arg_0], 41h
.text:00401064      jmp     short loc_40104D
.text:00401066 loc_401066:                  ; CODE XREF: sub_401040+1A?j
.text:00401066      mov     [ebp+arg_A], 0Ah
.text:0040106A      mov     [ebp+arg_C], 2710h
.text:00401071      mov     esp, ebp
.text:00401073      pop     ebp
.text:00401074      retn
.text:00401074 sub_401040  endp
```

Итак, рассмотрим код функции `init` (см. листинг 3.22). В принципе, текст нам уже вполне знаком. Это почти в точности код из листинга 3.19, который мы уже разбирали. Но вот здесь с учетом нашего анализа функции `main` мы уже можем кое-что понять. В функцию было отправлено 16 байтов (четыре раза по четыре байта), а в самой функции вначале обрабатывается массив из 10 байтов, затем `arg_0`, затем однобайтовая величина `arg_A`, а далее четырехбайтовая величина `arg_C`. И вот здесь уже вполне естественно предположить, что перед нами все-таки структура. Из чего это следует? Да из того, например, что в стек были отправлены три независимых (на первый

взгляд) двойных слова, а в процедуре первые 10 байтов оказываются связанными в один массив.

Итак, можно сделать вполне законный вывод: структуры проявляют себя при передаче их в качестве параметров. Однако, согласитесь, что наши рассуждения все же слишком эвристичны, чтобы поручать их дизассемблеру. Интересно, что компилятор Borland C++ действует в подобных случаях приблизительно так же, как Visual C++. Вот дизассемблерный фрагмент той же самой программы (см. листинг 3.21), который осуществляет вызов функции `init`.

```
.text:00401108    mov     al, byte_40C2C6
.text:0040110E    shl     eax, 10h
.text:00401111    mov     ax, word_40C2C4
.text:00401118    push    eax
.text:00401119    push    dword_40C2C0
.text:0040111F    push    dword_40C2BC
.text:00401125    push    dword_40C2B8
.text:0040112B    call    sub_401134
```

Фрагмент примечателен странной переменной `word_40C2C4`. Откуда вообще переменная типа `WORD` могла взяться? Ее нет в программе. Впрочем, общий объем передаваемых через стек данных опять составляет 16 байтов. Точнее, все же 15 байтов — Borland несколько аккуратнее? Вряд ли.

Однако существуют ситуации, когда дизассемблер (и мы вместе с ним) можем однозначно определить, что перед нами структура. Речь идет о ситуациях, когда структура используется в качестве параметров (опять параметров) при вызове известных библиотечных или API-функций. Следующий фрагмент (листинг 3.23) демонстрирует вызов API-функции `RegisterClass`. Я специально оставляю предшествующие вызову строки, где заполняется структура `WndClass`, которую дизассемблер прекрасно распознает. Да и как не распознать, если адрес этой структуры является параметром известной API-функции.

Листинг 3.23

```
.text:0040104D   mov    [ebp+WndClass.style], 0
.text:00401054   mov    [ebp+WndClass.lpfnWndProc], offset sub_401140
.text:0040105B   mov    [ebp+WndClass.cbClsExtra], 0
.text:00401062   mov    [ebp+WndClass.cbWndExtra], 0
.text:00401069   mov    edx, [ebp+hInstance]
.text:0040106C   mov    [ebp+WndClass.hInstance], edx
.text:0040106F   push   7F00h             ; lpIconName
```

```
.text:00401074    mov   eax, [ebp+hInstance]
.text:00401077    push  eax              ; hInstance
.text:00401078    call  ds:LoadIconA
.text:0040107E    mov   [ebp+WndClass.hIcon], eax
.text:00401081    push  7F00h            ; lpCursorName
.text:00401086    push  0                ; hInstance
.text:00401088    call  ds:LoadCursorA
.text:0040108E    mov   [ebp+WndClass.hCursor], eax
.text:00401091    mov   [ebp+WndClass.hbrBackground], 6
.text:00401098    mov   [ebp+WndClass.lpszMenuName], 0
.text:0040109F    lea   ecx, [ebp+ClassName]
.text:004010A2    mov   [ebp+WndClass.lpszClassName], ecx
.text:004010A5    lea   edx, [ebp+WndClass]
.text:004010A8    push  edx              ; lpWndClass
.text:004010A9    call  ds:RegisterClassA
```

В листинге 3.23 адрес структуры `WndClass` отсчитывается относительно содержимого регистра `EBP`, это значит, что структура определена как стековая локальная переменная *(см. разд. 3.1.2)*, но суть наших рассуждений не изменится, если сделать ее глобальной переменной, структура распознается по ее использованию в качестве параметра.

3.1.2. Локальные переменные

Обычно под *локальной переменной* понимают переменную, определенную непосредственно в процедуре или функции. Как известно, для этой цели используется стек. Но это частный случай. Под локальной переменной подразумевают более широкое понятие. Это не только переменные, определенные в стеке (их бы можно назвать стековыми), но и просто временные переменные (локальные во времени, по отношению ко времени выполнения программы), а также переменные, хранящиеся в регистрах. Наконец, параметры, передаваемые через стек, также можно рассматривать как локальные переменные.

Переменные, определенные в стеке

С переменными, определенными в стеке, мы уже неоднократно встречались. Я не строю свою книгу, как аксиоматическую теорию, поэтому многие понятия многократно появляются в книге, прежде чем о них будет рассказано систематично[5]. Считаю это нормальной и даже нужной практикой в препо-

[5] В программировании это называется "ссылка вперед".

давании и в учебных книгах. Это интригует, заставляет работать мозг и ждать, когда же автор (или лектор) соизволит, наконец, изложить данный вопрос.

В листинге 3.24 действуют только локальные переменные и две функции: `main` и `add`. Обратите внимание, что в функцию `add` передаются три параметра, причем первый параметр является указателем. Это не случайно, т. к. переменная `s` в функции `add` модифицируется.

Листинг 3.24

```
#include <stdio.h>
int add(int *, int, int);
void main()
{
  int i=10,s,j;
  s=12; j=20;
  printf("%d\n",add(&s,i,j));
};
int add(int * s1, int i1, int j1)
{
  int n;
  *s1=*s1+10;
  n=*s1+j1+i1;
  `    return n*n;
};
```

В листинге 3.25 представлен дизассемблированный текст функции `main` из листинга 3.24. Подчеркну, что мы установили опцию "запретить оптимизацию".

Листинг 3.25

```
.text:00401000 _main    proc near    ; CODE XREF: start+16E?p
.text:00401000    var_C  = dword ptr -0Ch
.text:00401000    var_8  = dword ptr -8
.text:00401000    var_4  = dword ptr -4
.text:00401000      push     ebp
.text:00401001      mov      ebp, esp
.text:00401003      sub      esp, 0Ch
.text:00401006      mov      [ebp+var_4], 0Ah
```

```
.text:0040100D       mov     [ebp+var_8], 0Ch
.text:00401014       mov     [ebp+var_C], 14h
.text:0040101B       mov     eax, [ebp+var_C]
.text:0040101E       push    eax
.text:0040101F       mov     ecx, [ebp+var_4]
.text:00401022       push    ecx
.text:00401023       lea     edx, [ebp+var_8]
.text:00401026       push    edx
.text:00401027       call    sub_401050
.text:0040102C       add     esp, 0Ch
.text:0040102F       push    eax
.text:00401030       push    offset unk_4060FC
.text:00401035       call    _printf
.text:0040103A       add     esp, 8
.text:0040103D       xor     eax, eax
.text:0040103F       mov     esp, ebp
.text:00401041       pop     ebp
.text:00401042       retn
.text:00401042 _main    endp
```

Комментарий к листингу 3.25

- Пропуская стандартный пролог функции, обращаем внимание на команду sub esp,0CH. Двенадцать байтов резервируется для локальных переменных — область между старым значением указателя стека (куда указывает регистр EBP) и новым значением. Это как раз три переменные (см. листинг 3.24). Впрочем, IDA Pro объявляет эти переменные как var_4, var_8, var_C. Что означают суффиксы _4, _8, _C? Это адреса, где располагаются переменные по отношению к границе, откуда начинается область стековых переменных. Адрес этой границы или начала области стековых переменных хранится в регистре EBP.
- Далее идут команды инициализации переменных. Обратите внимание, что нет разницы между переменной, которая инициализирована при объявлении, и переменными, которым просто присвоены значения в тексте программы.
- Адреса с 0040101B по 00401026 заняты командами, которые отправляют параметры в стек для вызова функции add. Обратим внимание на переменную var_8. Она, несомненно, обозначает переменную s в тексте программы. Для нее выполняются команды lea edx,[ebp+var_8]/push edx, т. е. в стек отправляется адрес этой переменной. Конечно, ведь в программе и указано, что передается указатель. Однако я хочу предосте-

речь вас от скороспелых выводов. Компиляторы очень часто обходятся с указателями довольно бесцеремонно. Дело в том, что тот факт, что у переменной s в функцию передается именно указатель, используется в программе для модификации переменной s. Если бы этого не было (не было модификации s в функции add), компилятор вполне мог передать в функцию саму переменную — хлопот меньше, а результат тот же. Итак, две другие переменные — i (var_4) и j (var_C) — передаются в стек как значения.

- Результат вызова функции, а он, как и следовало ожидать, хранится в регистре EAX *(см. разд. 3.2.1)*, передается в функцию в качестве параметра для вывода на консольный экран.

Теперь пришла пора разобрать код функции add. Он представлен в листинге 3.26.

Листинг 3.26

```
.text:00401050 sub_401050    proc near  ; CODE XREF: _main+27?p
.text:00401050    var_4  = dword ptr -4
.text:00401050    arg_0  = dword ptr  8
.text:00401050    arg_4  = dword ptr  0Ch
.text:00401050    arg_8  = dword ptr  10h
.text:00401050      push   ebp
.text:00401051      mov    ebp, esp
.text:00401053      push   ecx
.text:00401054      mov    eax, [ebp+arg_0]
.text:00401057      mov    ecx, [eax]
.text:00401059      add    ecx, 0Ah
.text:0040105C      mov    edx, [ebp+arg_0]
.text:0040105F      mov    [edx], ecx
.text:00401061      mov    eax, [ebp+arg_0]
.text:00401064      mov    ecx, [eax]
.text:00401066      add    ecx, [ebp+arg_8]
.text:00401069      add    ecx, [ebp+arg_4]
.text:0040106C      mov    [ebp+var_4], ecx
.text:0040106F      mov    eax, [ebp+var_4]
.text:00401072      imul   eax, [ebp+var_4]
.text:00401076      mov    esp, ebp
.text:00401078      pop    ebp
.text:00401079      retn
.text:00401079 sub_401050   endp
```

Комментарий к листингу 3.26

- IDA Pro для параметров функции дает имена, начинающиеся с префикса `arg`. Итак, как и должно, функция получила три параметра: `arg_0`, `arg_4`, `arg_8`. Смещения 0, 4, 8, как и в случае стековых переменных, отсчитываются от содержимого регистра `EBP`, но вниз, в область старших адресов.
- Обратите внимание, что здесь на первый взгляд не резервируется область в стеке для переменной `var_4` (в программе имя переменной — `n`). Это интересный момент. А почему же в функции `main` компилятор резервирует область в стеке под переменные? А дело здесь в том, что для резервирования области стека используется команда `push ecx`. Это легко понять, подсчитав баланс стека в начале и конце процедуры — количество байтов, положенных в стек в начале и извлеченных из стека в конце. Да, часто, когда стековая переменная одна, для резервирования стека используется именно команда `PUSH`.
- Интересно отыскать среди параметров функции тот, который представляет собой указатель на переменную. Здесь все просто. Его положили последним, а поскольку стек растет в сторону меньших адресов, то он будет иметь меньшее смещение в сторону старших адресов. Другими словами, это `arg_0`. Так оно и есть. Вот последовательность команд, которая выдает его с головой:

  ```
  mov eax,[ebp+arg_0]
  mov ecx,[eax]
  add ecx,0Ah
  ```

  это просто `*s1=*s1+10`.
- Дальнейшие выкладки достаточно очевидны и выражают просто действие `n=*s1+j1+i1`. Инструкция же `imul` — это действие `n*n`.

Нелишне опять напомнить об оптимизации. Она, особенно это касается Visual C++, может изменить исходный текст программы до неузнаваемости. Попробуем откомпилировать исходный текст программы (листинг 3.24) с опцией "создавать компактный код". Предварительно в функцию `add` следует вставить какой-нибудь оператор вывода, например, `printf("%d\n",n)`, иначе оптимизатор вообще обойдется без вызова функции и заменит ее вычисленной им самим константой (вот так!!![6]). А вот что получится с функцией `main` после того, как по ней пройдется оптимизатор (листинг 3.27).

---

[6] При оптимизации "максимальная скорость" и это не спасает!

Листинг 3.27

```
.text:00401029 _main    proc near    ; CODE XREF: start+16E?p
.text:00401029    var_4  = dword ptr -4
.text:00401029      push    ebp
.text:0040102A      mov     ebp, esp
.text:0040102C      push    ecx
.text:0040102D      push    14h
.text:0040102F      lea     eax, [ebp+var_4]
.text:00401032      push    0Ah
.text:00401034      push    eax
.text:00401035      mov     [ebp+var_4], 0Ch
.text:0040103C      call    sub_401000
.text:00401041      push    eax
.text:00401042      push    offset unk_4060FC
.text:00401047      call    _printf
.text:0040104C      add     esp, 14h
.text:0040104F      xor     eax, eax
.text:00401051      leave
.text:00401052      retn
.text:00401052 _main    endp
```

Комментарий к листингу 3.27

- Листинг весьма интересен и поучителен. Главное, на что следовало бы обратить внимание при анализе, — это то, что определена только одна стековая переменная. Какая эта переменная, надо бы догадаться, даже не глядя на текст листинга. Разумеется, это `s`. Именно ее содержимое будет модифицировано в функции `add`. Другими словами, `s` — это действительно переменная. Переменные же `i` и `j` — по сути, константы, т. к. в процессе работы программы не модифицируются. Так оптимизатор с ними и поступает. Вместо того чтобы выделять в стеке для них память, можно просто отправить в качестве параметров функции `add` числовые константы (команды `push 14h` и `push 0Ah`). Что касается переменной `s`, то в стек отправляется ее адрес:

  ```
  lea eax,[ebp+var_4]
  ...
  push eax
  ```

- Еще один интереснейший момент: память для стековой переменной отводится с помощью команды `push ecx`, что, конечно, может сбить с тол-

ку. Но перед оптимизатором поставлена задача сократить размер кода, и он старается изо всех сил. Этим же объясняется, что восстановление стека в конце процедуры осуществляется всего одной командой `leave`.

Итак, следующий вывод по стековым переменным: если в процессе выполнения программы содержимое стековой переменной не меняется, то оптимизатор может заменить ее константой. Конечно, для простого анализа того, что делает программа, данная информация, в принципе, не имеет большого значения. Но, мне кажется, для более глубокого понимания логики работы программы это важно.

Еще одну весьма полезную информацию можно почерпнуть, если откомпилировать пример из листинга 3.24 с помощью компилятора Borland C++ 5.0. Результат дизассемблирования функции `main` представлен в листинге 3.28.

Листинг 3.28

```
.text:00401108 _main    proc near   ; DATA XREF: .data:0040A0B8?o
.text:00401108    var_4 = dword ptr -4
.text:00401108    argc  = dword ptr  0Ch
.text:00401108    argv  = dword ptr  10h
.text:00401108    envp  = dword ptr  14h
.text:00401108      push  ebx
.text:00401109      push  esi
.text:0040110A      push  ecx
.text:0040110B      mov   ebx, 0Ah
.text:00401110      mov   [esp+4+var_4], 0Ch
.text:00401117      mov   esi, 14h
.text:0040111C      push  esi
.text:0040111D      push  ebx
.text:0040111E      lea   eax, [esp+0Ch+var_4]
.text:00401122      push  eax
.text:00401123      call  sub_401140
.text:00401128      add   esp, 0Ch
.text:0040112B      push  eax
.text:0040112C      push  offset format    ; format
.text:00401131      call  _printf
.text:00401136      add   esp, 8
.text:00401139      pop   edx
.text:0040113A      pop   esi
```

```
.text:0040113B        pop   ebx
.text:0040113C        retn
.text:0040113C  _main       endp
```

Комментарий к листингу 3.28

- Разные компиляторы — разные стили. Посмотрите, если компилятор Microsoft даже при объявлении, что функция `main` имеет тип `void`, обнуляет на всякий случай регистр `EAX`, то компилятор Borland понимает тип `void` буквально, т. е. не задумывается о содержимом `EAX`. Еще одна особенность: компилятор Borland во всю использует регистры `ESI` и `EBX`, а согласно принятым соглашениям функция не должна менять регистры `EBX`, `EBP`, `ESP`, `ESI`, `EDI` так, что ему приходится вставлять команды `PUSH EBX`/`PUSH ESI` в начале и `POP ESI`/`POP EBX` в конце функции. Я подозреваю, что это всего лишь рудимент прошлого. Дело в том, что в старых версиях Intel-регистры `CX` и `DX` не могли использоваться для адресации.
- Компилятор Borland, как и компилятор Microsoft, проанализировав текст, убеждается, что переменные `i` и `j` — по сути, константы, поэтому он не резервирует для них память в стеке, а использует просто константы. Резервируется память только для переменной `s` (`var_4`). Причем это производится также с помощью одной команды `PUSH` (`push ecx`).
- А вот теперь об интересном. Регистр `EBP` здесь вообще не используется, вместо него взят регистр `ESP`. Да-да, это известный оптимизирующий прием, и вам, дорогие читатели, необходимо о нем знать и помнить. "Но ведь содержимое-то регистра `ESP` меняется", — скажете вы и будете, конечно, правы. Но компилятор не такой простак, чтобы забыть об этом, и прекрасно справляется с данной проблемой, динамически отслеживая изменения регистра `ESP` и подстраивая адресацию. Смотрите, в начале была команда `mov [esp+4+var_4],0Ch`. Затем последовали две команды `PUSH`, т. е. содержимое `ESP` уменьшилось на 8. Поэтому далее компилятор пишет `lea eax,[esp+0Ch+var_4]`, все верно: 4 + 8 = 12 = `0Ch`. Кстати IDA Pro, к счастью, эти вещи также понимает и в обеих командах указывает переменную `var_4`.

Временные переменные

Что такое *временные переменные*? Я рассматриваю их как переменные, используемые для хранения промежуточных результатов вычислений. При вычислениях широко применяеются регистры процессора. Поэтому можно сказать, регистры используются в качестве временных переменных. Мы уже встречались с таким использованием. Возьмите хотя бы листинг 3.10 и организацию в нем цикла (адреса `00401027—0040102F`). Регистр `EAX` играет здесь как раз роль временной переменной, в которой хранится (временно, пока

работает цикл) параметр цикла. При использовании вещественных переменных для хранения промежуточных результатов подключаются регистры арифметического сопроцессора. Как правило, для этих целей служат три первых регистра арифметического сопроцессора: `ST(0)`, `ST(1)` и `ST(2)`. Если вернуться опять же к листингу 3.10, то там в комментарии к листингу я обращал ваше внимание на способ начальной инициализации вещественных переменных: вещественная константа вначале загружается в регистр `ST(0)` сопроцессора (команда `fld`), а затем оттуда загружается в область памяти, которая отведена для вещественной переменной (команда `fstp`).

Сколько же всего регистров может понадобиться, если вычисляемое выражение сложное? Но давайте просто порассуждаем. Действия, которые мы выполняем над числовыми переменными, являются бинарными действиями. Другими словами, в каждой операции участвуют два операнда. А результат можно поместить либо в третий операнд, либо в один из операндов, участвующих в предыдущем действии. Результат выполнения действия может быть операндом в другой бинарной операции, но опять в операции участвуют два операнда, а результат помещается в один. Эти рассуждения применимы и в том случае, если в выражении имеются скобки. Из данного рассуждения можно сделать вывод, что для хранения промежуточных результатов достаточно двух операндов. А как быть, если операнды имеют длину 64 бита (если процессор 32-битный)? Для этой цели компилятор C++ может использовать библиотечные процедуры (например, `_alldiv`), которые припасены на такой случай. Впрочем, как мы увидим далее, иногда компилятор все же использует стек под временные переменные.

Ну, стало быть, пора разобрать какой-нибудь поучительный пример. Лучше всего для этой цели подходит какое-нибудь вычисление. В листинге 3.29 есть такое вычисление, причем в формуле участвуют и целые, и вещественные типы переменных.

Листинг 3.29

```
#include <stdio.h>
void main()
{
  double i,j,s;
  int  k,d;
  i=10; j=20; k=30; d=40;
  s=((k-1)*(d-1))*((i-1)/(j-1));
  printf("%f\n",s);
};
```

В листинге 3.30 представлен дизассемблированный код функции `main`, взятый из дизассемблера IDA Pro.

Листинг 3.30

```
.text:00401000  _main    proc near    ; CODE XREF: start+16E?p
.text:00401000    var_2C  = qword ptr -2Ch
.text:00401000    var_24  = dword ptr -24h
.text:00401000    var_20  = qword ptr -20h
.text:00401000    var_18  = dword ptr -18h
.text:00401000    var_14  = dword ptr -14h
.text:00401000    var_10  = qword ptr -10h
.text:00401000    var_8   = qword ptr -8
.text:00401000      push    ebp
.text:00401001      mov     ebp, esp
.text:00401003      sub     esp, 24h
.text:00401006      fld     ds:dbl_408110
.text:0040100C      fstp    [ebp+var_8]
.text:0040100F      fld     ds:dbl_408108
.text:00401015      fstp    [ebp+var_20]
.text:00401018      mov     [ebp+var_14], 1Eh
.text:0040101F      mov     [ebp+var_18], 28h
.text:00401026      mov     eax, [ebp+var_14]
.text:00401029      sub     eax, 1
.text:0040102C      mov     ecx, [ebp+var_18]
.text:0040102F      sub     ecx, 1
.text:00401032      imul    eax, ecx
.text:00401035      mov     [ebp+var_24], eax
.text:00401038      fild    [ebp+var_24]
.text:0040103B      fld     [ebp+var_8]
.text:0040103E      fsub    ds:dbl_408100
.text:00401044      fld     [ebp+var_20]
.text:00401047      fsub    ds:dbl_408100
.text:0040104D      fdivp   st(1), st
.text:0040104F      fmulp   st(1), st
.text:00401051      fst     [ebp+var_10]
.text:00401054      sub     esp, 8
.text:00401057      fstp    [esp+2Ch+var_2C]
.text:0040105A      push    offset unk_4080FC
```

```
.text:0040105F      call    _printf
.text:00401064      add     esp, 0Ch
.text:00401067      xor     eax, eax
.text:00401069      mov     esp, ebp
.text:0040106B      pop     ebp
.text:0040106C      retn
.text:0040106C _main    endp
```

Комментарий к листингу 3.30

- На локальные переменные отводиться 36 байтов (`sub esp,24h`). Это больше на четыре байта, чем требуется для пяти переменных. Компилятор не удержался и выделил стек для временной переменной, хотя, на первый взгляд, мог бы этого и не делать, т. к. можно было бы задействовать резерв в лице регистра EDX или оставить в регистре EAX (см. далее). Компилятор Microsoft избегает использовать для вычислений регистры EBX, EDI, ESI, т. к. пришлось бы принимать меры для их восстановления в конце функции.
- По адресам 00401006—0040101F стоят команды начальной инициализации переменных. Для инициализации вещественных переменных, как и ранее, компилятор использует вещественные константы[7], хранящиеся в сегменте данных. При этом константа вначале загружается в регистр ST(0) арифметического сопроцессора (команда `fld`), а потом уже в переменную (команда `fstp`). Целые переменные инициализируются непосредственной загрузкой в них (команда `mov`) определенных значений.
- Далее начинаются непосредственные вычисления. Давайте подробно в них разберемся.
  - Команды 00401026—0040102F — это загрузка в регистры переменных `k` и `d` и дальнейшая подготовка их к выполнению умножения. Подготовка, собственно, сводится к вычитанию из них единицы. Таким образом, в EAX у нас оказалась разность `k-1`, а в ECX — разность `d-1`. Теперь можно производить умножение. Далее — команда `imul eax,ecx`, и результат умножения оказывается в регистре EAX. Другими словами, `(k-1)*(d-1)->EAX`. И вот теперь надо решать вопрос, где хранить результат этих вычислений. Регистр EAX вполне бы подошел, т. к. далее он, вроде бы, более не используется в вычислениях. Но есть одно "но". Получившийся целый результат потом должен участвовать в вычислениях с вещественными числами. Команда же `fld` загружает стек

---

[7] В языке C++ константы, хранящиеся в сегменте данных и имеющие, как и переменная, определенный тип, называют *типизированными*. Константы, которые используются только непосредственно в тексте программы, называют *литеральными*.

арифметического сопроцессора из области памяти. Таким образом, компилятор вполне разумно решил, что есть смысл использовать временную переменную для хранения промежуточного результата прямо в стеке.

- Перейдем теперь к дальнейшим вычислениям. Итак, далее результат вычисления выражения (k-1)*(d-1) с помощью команды fild загружается в вершину стека арифметического сопроцессора, т. е. в регистр ST(0). Затем следует команда fld, загружающая в ST(0) переменную i. При этом старое значение ST(0) перемещается в ST(1). Потом команда fsub ds:dbl_408100 (адрес 0040103E) вычисляет i-1. Результат при этом остается в ST(0). Следующая команда fld загружает в ST(0) переменную j. При этом (будьте внимательны!) старое содержимое ST(0) перемещается в ST(1), а старое содержимое ST(1) перемещается в ST(2). Таким образом, ST(2) играет здесь роль временной переменной. Следующая команда fsub вычисляет j-1. Теперь fdivp st(1),st — это деление с выталкиванием из стека. В результате частное оказывается в ST(0), а то, что было в ST(2), переходит в ST(1). Команда fmulp st(1),st — это умножение с выталкиванием из стека. Таким образом, окончательный результат оказывается в ST(0). А вот это уже последний штрих: fst [ebp+var_10], что просто означает ST(0)->s. Заметим, что команда fst помещает значение в переменную без выталкивания из стека.
- Далее для отправки в стек вещественного числа используется уже известный нам прием: команда sub esp,8, равносильная двум командам PUSH, готовит место для вещественной переменной. И далее команда fstp (уже с выталкиванием из стека сопроцессора) помещает результат вычислений в стек для использования в функции printf.

Итак, временные переменные используются компилятором для проведения каких-либо выкладок. В качестве временных переменных могут выступать обычные регистры, регистры арифметического сопроцессора, а также стековые переменные.

Очень часто временные переменные нужны, когда результаты выполнения одной функции используются в другой функции.

В листинге 3.31 представлена программа, в которой результат выполнения функции sub используется в функции add, а результат выполнения функции add, в свою очередь, используется в printf. В листинге 3.32 представлен фрагмент дизассемблированного IDA Pro кода, как раз касающийся временных переменных.

Листинг 3.31

```
#include <stdio.h>
int add(int, int);
int sub(int, int);
void main()
{
  int i=10,j=20;
  printf("%d\n",add(i,sub(i,j)));
};

int add(int a, int b)
{
  return a+b;
};
int sub(int a, int b)
{
  return a-b;
};
```

Листинг 3.32

```
.text:00401014    mov     eax, [ebp+var_8]
.text:00401017    push    eax
.text:00401018    mov     ecx, [ebp+var_4]
.text:0040101B    push    ecx
.text:0040101C    call    sub_401060
.text:00401021    add     esp, 8
.text:00401024    push    eax
.text:00401025    mov     edx, [ebp+var_4]
.text:00401028    push    edx
.text:00401029    call    sub_401050
.text:0040102E    add     esp, 8
.text:00401031    push    eax
.text:00401032    push    offset unk_4060FC
.text:00401037    call    _printf
.text:0040103C    add     esp, 8
```

Комментарий к листингу 3.32

Переменные var_4 и var_8 соответствуют i и j в программе. Итак, вначале вызывается функция sub_401060, т. е. sub. Результат функции, как и следовало, оказывается в регистре EAX. Потом регистр EAX используется уже как переменная, которая затем обрабатывается как параметр при вызове функции add (sub_401050). Далее аналогично: результат опять оказывается в регистре EAX, и он (регистр) используется в качестве параметра вызова функции printf.

Регистровые переменные

В языке С предусмотрен тип переменной register. Изначально предполагалось, что переменные, определенные, как register, должны, по возможности, храниться именно в регистре. Современные компиляторы не обращают внимания на это ключевое слово (хотя для совместимости и признают его), а действуют так, как подсказывают им соображения целесообразности, а также установленные опции оптимизации. Рассмотрим простую программу, представленную в листинге 3.33. Откомпилируем программу с помощью компилятора Visual C++ с опцией "создавать компактный код".

Листинг 3.33

```
#include <stdio.h>
void main()
{
  int i,j,s;
  i=0; j=1; s=0;
  for(i=0; i<100; i++,j++)s=s+j;
  printf("%d %d %d \n",i,j,s);
};
```

Результат дизассемблирования исполняемого кода программы из листинга 3.33 представлен в листинге 3.34.

Листинг 3.34

```
.text:00401000  _main     proc near           ; CODE XREF: start+16E?p
.text:00401000          xor     eax, eax
.text:00401002          push    64h
.text:00401004          inc     eax
.text:00401005          xor     ecx, ecx
```

```
.text:00401007      pop     edx
.text:00401008  loc_401008:                    ; CODE XREF: _main+C?j
.text:00401008      add     ecx, eax
.text:0040100A      inc     eax
.text:0040100B      dec     edx
.text:0040100C      jnz     short loc_401008
.text:0040100E      push    ecx
.text:0040100F      push    eax
.text:00401010      push    64h
.text:00401012      push    offset aDDD      ; "%d %d %d \n"
.text:00401017      call    _printf
.text:0040101C      add     esp, 10h
.text:0040101F      xor     eax, eax
.text:00401021      retn
.text:00401021  _main    endp
```

Комментарий к листингу 3.34

- Обратите внимание, что, хотя в исходной программе определены три локальные переменные, в результирующем коде стек для хранения переменных не используется. И это как раз тот случай, когда компилятор воспользовался регистрами для хранения переменных. Замечу также, что в целях экономии размера кода в функции `main` отсутствуют пролог и эпилог.
- Итак, регистр `ECX` используется для хранения переменной `s` (`xor ecx,ecx` — это просто `s=0`). Далее команды

  ```
  xor eax,eax
  ...
  inc eax
  ```

  относятся к переменной `j`. Что касается переменной `i`, то здесь сделана довольно интересная модификация, в целях уменьшения кода разумеется. Вместо того чтобы увеличивать значение некоторой переменной, а затем сравнивать ее со значением 100, некоторой переменной присваивается значение 100, и после каждого прохода тела цикла значение переменной уменьшается, а получившийся результат сравнивается с 0. Это и короче, и быстрее. В качестве такой регистровой переменной выступает `EDX`.
- Наконец, последнее. Поскольку в конце выполнения цикла у нас нет переменной, содержащей значение 100 (как должно быть по программе), в стек отправляется просто число 100 (`push 64h`).

3.2. Идентификация программных структур

Понимание программной структуры исполняемого модуля часто важнее распознавания переменных, т. к. позволяет понять логику выполнения программы.

3.2.1. Процедуры и функции

С процедурами и функциями[8] мы уже встречались неоднократно. Сейчас постараемся обобщить уже имеющийся у нас опыт и исследовать новые особенности.

Передача параметров

До сих пор мы молчаливо предполагали, что данные в процедуру передаются через стек. Да, этот механизм используется чаще всего, и об этом мы будем говорить в следующем разделе. Но данный механизм — отнюдь не единственный.

Отвлечемся пока от компиляторов и просто поразмышляем над тем, как и каким образом, в принципе, могут передаваться параметры в процедуру. Если вы работаете на ассемблере, то все перечисленные ниже механизмы могут быть взяты на вооружение. Боле того, никто не помешает вам использовать сразу несколько механизмов. При работе с компиляторами языков высокого уровня приходится считаться с принятыми соглашениями, но о них будет сказано позднее.

Через стек

Передача параметров через стек является самым распространенным способом. Он позволяет создавать *рекурсивные процедуры*, тогда как использование других подходов делает рекурсию весьма проблематичной. Параметры помещают в стек обычно при помощи команд PUSH. Но возможен и другой способ, с которым мы неоднократно встречались. Можно вручную изменить значение указателя стека, а затем с помощью обычных команд MOV поместить параметры в выделенную область. Например, если два параметра находятся, соответственно, в регистрах EAX и EBX, то поместить их в стек можно последовательностью команд:

```
SUB ESP,8
MOV [ESP],EAX
MOV [ESP],EBX
```

---

[8] В языках высокого уровня принято различать процедуры и функции. С точки же зрения дизассемблированного текста, разницы между этими двумя понятиями нет.

Это равносильно двум командам:

```
PUSH EAX
PUSH EBX
```

Напоминаю, что стек растет вверх в сторону меньших адресов.

При передаче параметров важным вопросом является порядок следования параметров в стеке. Вызываемая процедура при взятии параметров из стека следует вполне определенному порядку, этого порядка и необходимо придерживаться при вызове данной процедуры. Но это лишь одна проблема. Вторая проблема заключается в освобождении стека. После того как вызываемая процедура выполнила предполагаемые действия и возвратила управление в вызывающий ее фрагмент программы, в стеке остаются передаваемые параметры. Многократный вызов процедуры, в конце концов, может привести программу к краху. Практикуются два подхода к решению этой проблемы. Первый способ применяется в языке C++. Он заключается в том, что стек освобождается уже после того, как осуществлен возврат из данной процедуры. Это весьма удобно, поскольку тогда можно использовать процедуры с переменным числом параметров.

Замечание

Примером такой процедуры может служить стандартная библиотечная функция C `printf`. Первым параметром этой функции всегда идет строка, которая может содержать специальные подстроки (они называются *спецификаторами формата*), начинающиеся с символа `%`. Количество таких подстрок равно количеству дополнительных параметров функции `printf`.

Для восстановления стека используется обычно команда `ADD ESP,4*N`, где *N* — количество 32-битных параметров[9]. Но возможна и такая команда: `SUB ESP,-4*N`, или даже команды `POP`. Важно понимать их назначение. Иногда компилятор может в целях экономии восстанавливать стек, так сказать, оптом, после вызова сразу нескольких процедур.

Второй способ восстановления стека заключается в использовании при выходе из процедуры команды `RETN 4*N`, здесь *N* — опять же количество 32-битных параметров. Такой подход изначально использовался в компиляторах языка Паскаль. Конечно, это несколько быстрее, но зато проблематично вызывать процедуру с переменным количеством параметров.

Через сегмент данных

Данный подход весьма очевиден. Использование глобальных переменных для передачи информации в процедуру просто напрашивается. Однако такой подход вызывает и головную боль. Действительно, чтобы не наделать

[9] Параметр типа `double` следует, очевидно, расценивать как два 32-битных.

ошибок, вам придется для каждой процедуры выделить свой набор глобальных переменных, а это уже расход памяти. С другой стороны, использование глобальных переменных делает проблематичным рекурсивный вызов: вы же не можете использовать те же переменные, если они уже используются. Однако этот недостаток совсем не означает, что данный подход не используется. Ничто не мешает вам применять его при написании программы на C++ или Delphi.

Описанный выше подход можно усовершенствовать, если использовать для передачи параметров через специально организованный блок памяти. Скорее всего, такой блок вам придется организовывать для каждой процедуры, хотя теоретически можно придумать структуру универсального буфера, через который будут передаваться параметры для всех используемых процедур. Структура такого буфера может быть организована таким образом, что станет возможным рекурсивный вызов процедур.

В программном коде

Передача параметров в программном коде весьма экзотична, но вполне реальна, если воспользоваться ассемблером. Рассмотрим следующую схему:

```
...
CALL PROC1
  DB "Этот параметр передается в программном коде",0
;сюда будет осуществляться возврат из процедуры PROC1
...

PROC1 PROC
;извлекаем из стека адрес возврата
;определяем адрес параметров и их длину
;изменяем адрес возврата в стеке
;обработка
;возврат из процедуры
  RETN
PROC1 ENDP
```

Как видим, в этой схеме нет ничего сложного или невероятного. Однако реализация его на языке высокого уровня требует дополнительных усилий.

При помощи регистров

Передача параметры через регистры является довольно быстрым способом. Однако есть, разумеется, и ограничения, поскольку количество регистров не так уж и велико. Такой подход применяется в основном в сочетании с другими механизмами, обычно стековым способом передачи параметров —

первые параметры передаются через регистры, а остальные — через стек. Мы будем говорить об этом подходе далее.

Соглашения о передаче параметров

Обратимся теперь к компиляторам языков высокого уровня. Как и следовало ожидать, они в основном используют стековый способ передачи параметров. Таблица 3.2 содержит сведения об основных соглашениях, используемых современными компиляторами.

Таблица 3.2. *Стандартные соглашения о передаче параметров*

Название соглашения	Порядок следования параметров	Способ освобождения стека	Комментарий
Си-соглашение (__cdecl)	Справа налево	Вызывающая программа	Компилятор автоматически подставляет перед именем функции знак подчеркивания (_)
Стандартное соглашение (__stdcall)	Справа налево	Вызываемая процедура	Компилятор автоматически подставляет перед именем функции знак подчеркивания (_). В конце имени функции ставится суффикс @, а за ним идет число, определяющее суммарную длину параметров в байтах
Соглашение языка Паскаль (__pascal)	Слева направо	Вызываемая процедура	Используется в языках Паскаль и Delphi
Соглашение быстрого вызова (__fastcall). Это соглашение еще называют *регистровым вызовом*	Слева направо	Вызываемая процедура	Для компилятора Microsoft C++ задействованы два регистра (ECX, EDX). Если для передачи параметров их не хватает, то остальные параметры передаются через стек. Компилятор Borland C++ использует три регистра (EAX, EDX, ECX)

Замечание

Представленные в табл. 3.2 соглашения не единственные. В разных языках есть свои соглашения. Например, Delphi поддерживает соглашение __safecall, Basic имеет свое соглашение. Некоторые соглашения выходят из употребления. Так, соглашение языка Паскаль (__pascal) не поддерживается больше Microsoft Visual C++.

При написании программ на C++ нам чаще всего встречаются соглашения __cdecl (при работе с обычными и библиотечными функциями) и __stdcall (при вызове большинства API-функций).

В качестве иллюстрации использования регистрового соглашении, т. е. быстрого вызова функций рассмотрим следующую простую программу (листинг 3.35).

Листинг 3.35

```
#include <stdio.h>
int __fastcall  add(int, int , int);
void main()
{
  int i=10,j=20,k=30;
  printf("%d\n",add(i,j,k));
};

int __fastcall add(int a, int b, int c)
{
  return a+b+c;
};
```

Как видим, в программе имеется функция, объявленная как __fastcall. Рассмотрим вначале дизассемблер исполняемого кода, сделанного компилятором Microsoft C++ (листинг 3.36).

Листинг 3.36

```
.text:00401000  _main    proc near    ; CODE XREF: start+16E?p
.text:00401000    var_C  = dword ptr -0Ch
.text:00401000    var_8  = dword ptr -8
.text:00401000    var_4  = dword ptr -4
.text:00401000      push    ebp
.text:00401001      mov     ebp, esp
.text:00401003      sub     esp, 0Ch
.text:00401006      mov     [ebp+var_4], 0Ah
.text:0040100D      mov     [ebp+var_C], 14h
.text:00401014      mov     [ebp+var_8], 1Eh
.text:0040101B      mov     eax, [ebp+var_8]
.text:0040101E      push    eax
```

```
.text:0040101F      mov     edx, [ebp+var_C]
.text:00401022      mov     ecx, [ebp+var_4]
.text:00401025      call    sub_401040
.text:0040102A      push    eax
.text:0040102B      push    offset unk_4060FC
.text:00401030      call    _printf
.text:00401035      add     esp, 8
.text:00401038      xor     eax, eax
.text:0040103A      mov     esp, ebp
.text:0040103C      pop     ebp
.text:0040103D      retn
.text:0040103D  _main    endp
```

Комментарий к листингу 3.36

Код, представленный в листинге, для нас достаточно знаком. Но есть один момент, с которым мы еще не встречались. Согласно программе из листинга 3.36, у функции `add` должны быть три параметра. Очевидно, что `sub_401040` и есть функция `add`. Далее команды `mov eax, [ebp+var_8]/push eax` отправляют в стек последнюю переменную `k`. Значения же переменных `i` и `j` помещаются в регистры `ECX` и `EDX` соответственно. Но это и есть соглашение __fastcall, принятое для компилятора Visual C++. В документации компилятора указано, что он выполняет предписание быстрого вызова по мере возможности. Так оно и есть, если увеличить количество параметров, то компилятор будет передавать их уже обычным образом через стек. Это очень легко объяснить, ведь в процедуре, которая будет вызвана, регистры также нужны, и при увеличении количества параметров рабочих регистров не хватит, и придется создавать локальные стековые переменные.

В листинге 3.37 представлен дизассемблированный код, полученный с помощью компилятора Borland C++ из той же программы.

Листинг 3.37

```
.text:00401108  _main    proc near  ; DATA XREF: .data:0040A0B8?o
.text:00401108    argc    = dword ptr  10h
.text:00401108    argv    = dword ptr  14h
.text:00401108    envp    = dword ptr  18h
.text:00401108      push    ebx
.text:00401109      push    esi
.text:0040110A      push    edi
.text:0040110B      mov     ebx, 0Ah
.text:00401110      mov     esi, 14h
```

```
.text:00401115      mov     edi, 1Eh
.text:0040111A      mov     ecx, edi
.text:0040111C      mov     edx, esi
.text:0040111E      mov     eax, ebx
.text:00401120      call    sub_401138
.text:00401125      push    eax
.text:00401126      push    offset format   ; format
.text:0040112B      call    _printf
.text:00401130      add     esp, 8
.text:00401133      pop     edi
.text:00401134      pop     esi
.text:00401135      pop     ebx
.text:00401136      retn
.text:00401136 _main    endp
```

Комментарий к листингу 3.37

Как мы видим из листинга, компилятор Borland C++ посылает параметры последовательно в регистры EAX, EDX, ECX. Замечу, кстати, что компилятор Borland вместо стековых переменных использует регистровые переменные в регистрах EBX, ESI, EDI. В отличие о компилятора Microsoft, Borland относится к модификатору __fastcall "серьезно" и не отменяет предписание использовать регистры с увеличением количества параметров.

Структуры стека

Мы уже многократно изучали различные листинги, где обращали внимание на то, где в стеке располагаются адрес возврата, параметры, локальные и временные переменные. Сейчас наша задача — обобщить имеющийся опыт, а также получить некоторую новую информацию.

Итак, смотрим на рис. 3.2. На нем изображены стадии, которые проходит стек, при вызове процедуры. Процесс изменения стека начинается с вызова процедуры (стадии 1—3) с помещением в стек параметров и заканчивается выделением памяти для локальных переменных и сохранением в стеке регистров, которые будут использоваться в процедуре и значения которых не должны изменяться после вызова (стадии 4 и 5). Рассмотрим подробнее эти стадии.

1. Помещение параметров в стек осуществляется чаще всего командами PUSH *reg32* или PUSH DWORD PTR *mem*, где *reg32* — 32-битный регистр, *mem* — адрес области памяти (прямой или косвенный). Но возможен и другой способ отправки параметров в стек. Вначале в стеке выделяется область для параметров, например, так: SUB ESP,*N*, где *N* — количество

необходимых для параметров байтов, кратных 4. Затем с помощью обычных команд MOV параметры помещаются в стек.

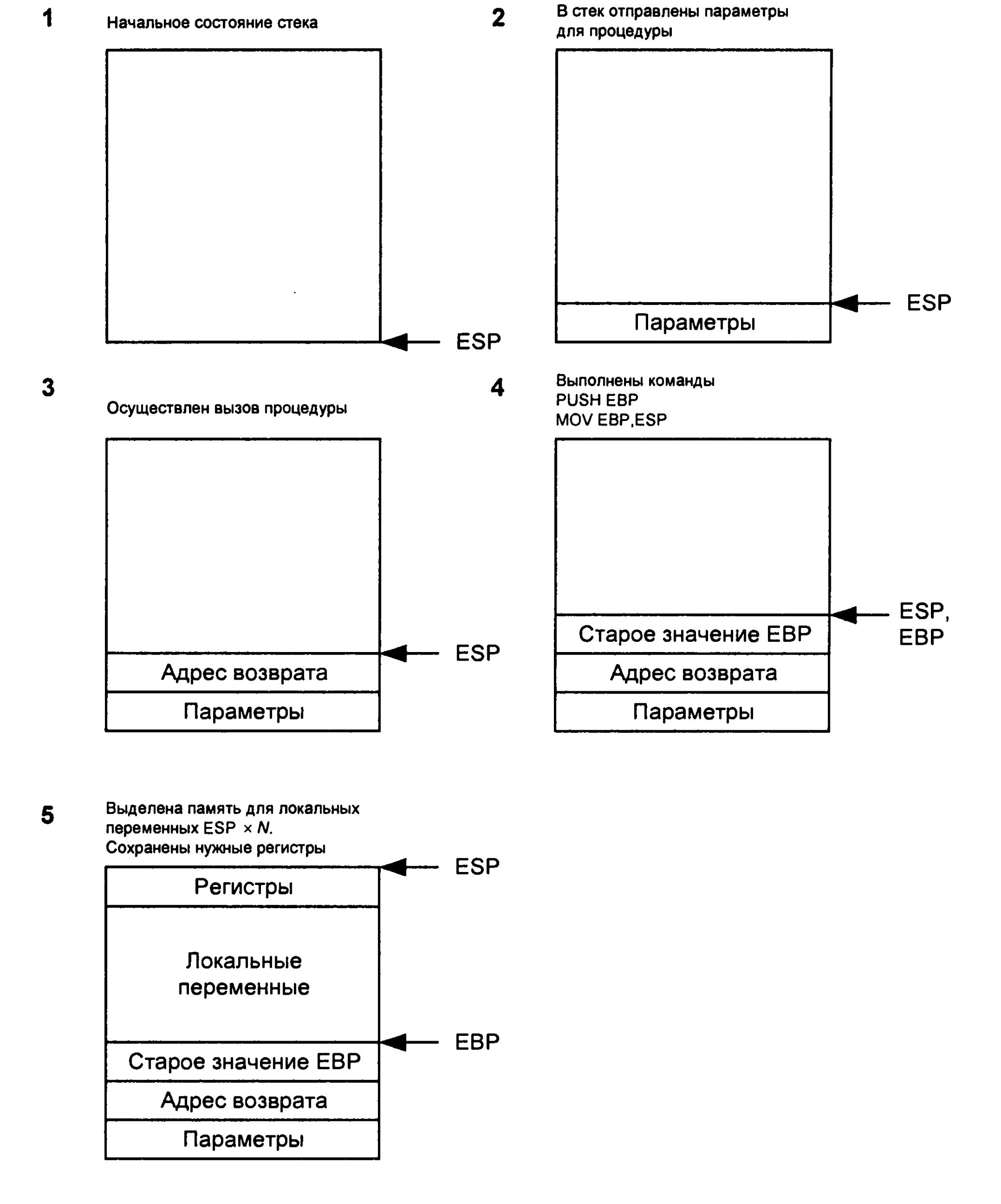


Рис. 3.2. Стандартная структура стека при вызове процедуры

Например, так:

```
MOV DWORD PTR [ESP],EAX
MOV DWORD PTR [ESP+4],EBX
```

и т. д. Если мы имеем дело с операндом типа double, имеющим размер 8 байтов, то для помещения его в стек используется команда FSTP, например, FSTP DWORD PTR [ESP], и 8 байтов из регистра арифметического сопроцессора ST(0) будет отправлено в стек (см. листинг 3.10 и комментарий к нему).

2. Команда CALL помещает в стек (за параметрами, если они есть) адрес возврата. Адрес возврата — это адрес следующей за командой CALL команды. Для того чтобы правильно возвратиться из процедуры, этот адрес должен находиться на вершине стека. Кроме этого, команда CALL осуществляет переход по указанному в ней адресу. Теперь работа по обустраиванию стека переносится в процедуру. Обычно процедура начинается с команды PUSH EBP. Эта команда сразу предполагает дальнейшее использование EBP, и, скорее всего, этот регистр будет нужен для адресации стековых переменных и параметров. Подтверждением тому будет следующая команда: MOV EBP,ESP. Для чего это делается? Дело в том, что регистр ESP привязан к командам PUSH и POP, которые изменяют его автоматически. Следовательно, если самый близкий к вершине стека параметр располагался в начале процедуры по адресу [ESP+4], то после команды PUSH он будет располагаться по адресу [ESP+8]. При помощи регистра EBP фиксируется точка отсчета для параметров и стековых переменных.

3. Следующий шаг в формировании структуры стека — это выделение области для хранения локальных переменных. Тут сразу надо оговориться, что если локальные переменные не предполагается использовать, то, соответственно, этот шаг компилятором пропускается. Выделение стека осуществляется обычно командой SUB ESP,*N*, где *N* — количество выделяемых байтов, кратное 4. Однако в некоторых случаях может быть использована команда ADD ESP,-*N* или несколько команда PUSH. Использование команды PUSH удобно с той точки зрения, что в одной команде можно совместить выделение стека и инициализацию переменной (см. листинг 3.26 и комментарий к нему). Последовательность команд

```
PUSH EBP
MOV EBP,ESP
SUB ESP,N
```

можно заменить всего одной командой ENTER *N*, которая, однако, почти совсем не используется компиляторами из-за своей чрезвычайной медлительности.

4. Наконец, если в процедуре предполагается использовать регистры EBX, ESI или EDI, они должны быть сохранены в стеке.
5. В конце процедуры состояние стека должно быть возвращено к состоянию, когда на вершине стека должен располагаться адрес возврата из процедуры. Кроме этого, должны быть восстановлены регистры EBP, EBX, ESI, EDI, если они, конечно, менялись. Довольно часто встречается последовательность команд

   ```
   MOV ESP,EBP
   POP EBP
   ```

   которую компилятор заменяет всего одной командой leave.

Если бы изложенная выше схема соблюдалась неукоснительно, то распознавание процедуры при дизассемблировании не составляло никакого труда, даже если бы процедура вызывалась с помощью косвенных команд вызова (CALL *reg32*, CALL [*reg32*], CALL [*mem*]). Современные компиляторы, однако, в целях оптимизации стали отказываться от использования регистра EBP для адресации стековых переменных и параметров (см. листинг 3.28 и комментарий к нему).

Очень интересным, на мой взгляд, является вопрос о вложенных процедурах. В C++ вложенные функции не возможны, а вот Паскаль вполне допускает возможность такого конструирования (листинг 3.38).

Листинг 3.38

```
program Project1;
var
a:integer;
procedure proc1(a1:integer);
var  b,g,d,e:integer;
  procedure proc2(a1:integer);
  var c:integer;
  begin
    c:=30;
    writeln(a1,b,c,d,e,g);
  end;
begin
   b:=20; g:=30; d:=40; e:=50;
   proc2(a1);
end;
```

```
begin
  a:=10;
  proc1(a);
end.
```

В листинге 3.39 представлена дизассемблированная основная (стартовая) часть откомпилированной в Delphi программы из листинга 3.38.

Листинг 3.39

```
CODE:004039B4      public start
CODE:004039B4   start:
CODE:004039B4      push    ebp
CODE:004039B5      mov     ebp, esp
CODE:004039B7      add     esp, 0FFFFFFF0h
CODE:004039BA      mov     eax, ds:off_4040A8
CODE:004039BF      mov     byte ptr [eax], 1
CODE:004039C2      mov     eax, offset dword_403994
CODE:004039C7      call    sub_403860
CODE:004039CC      mov     ds:dword_40565C, 0Ah
CODE:004039D6      mov     eax, ds:dword_40565C
CODE:004039DB      call    sub_403938
CODE:004039E0      call    sub_403394
```

Комментарий к листингу 3.39

В листинге 3.39 представлена стартовая часть программы. Из трех вызовов процедур, которые мы видим в листинге, один является вызовом процедуры, которая имеется непосредственно в прикладной программе (процедура `proc1`). Очевидно, это процедура `sub_403938`. Две другие процедуры — системные и выполняются при запуске программы (начальная инициализация) и при окончании работы программы. Процедура `sub_403938` получает свой единственный параметр через регистр `EAX`. Другими словами, в Delphi "процветает" вызов __fascall, хотя в программе мы его, вроде бы, и не заказывали. Я даже при компиляции отменил опцию оптимизации, но, как видите, Delphi управился по-своему. Переменная `dword_40565C` соответствует в программе переменной `a`, она и передается в процедуру через регистр. Еще прошу обратить внимание на команду `add esp,0FFFFFFF0h`. Я надеюсь, вам не составит труда сообразить, что перед нами в действительности команда `add esp,-16`, что, конечно, равносильно `sub esp,16`, т. е. резервируется 16 байтов.

В листинге 3.40 дан дизассемблированный текст откомпилированной процедуры `proc1` (`sub_403938`).

Листинг 3.40

```
CODE:00403938  sub_403938  proc near  ; CODE XREF: CODE:004039DB?p
CODE:00403938    var_14  = dword ptr -14h
CODE:00403938    var_10  = dword ptr -10h
CODE:00403938    var_C  = dword ptr -0Ch
CODE:00403938    var_8  = dword ptr -8
CODE:00403938    var_4  = dword ptr -4
CODE:00403938      push    ebp
CODE:00403939      mov     ebp, esp
CODE:0040393B      add     esp, 0FFFFFFECh
CODE:0040393E      mov     [ebp+var_14], eax
CODE:00403941      mov     [ebp+var_4], 14h
CODE:00403948      mov     [ebp+var_10], 1Eh
CODE:0040394F      mov     [ebp+var_8], 28h
CODE:00403956      mov     [ebp+var_C], 32h
CODE:0040395D      push    ebp
CODE:0040395E      mov     eax, [ebp+var_14]
CODE:00403961      call    sub_4038DC
CODE:00403966      pop     ecx
CODE:00403967      mov     esp, ebp
CODE:00403969      pop     ebp
CODE:0040396A      retn
CODE:0040396A  sub_403938  endp
```

Комментарий к листингу 3.40

- Обратим внимание на то, в процедуре `proc1` определены четыре локальные переменные. Однако мы видим, что в действительности в исполняемом коде определены пять локальных переменных. Переменная `var_14` отведена для хранения параметра, переданного в процедуру (`mov [ebp+var_14],eax`), т. е. является временной переменной. Команда `add esp,0FFFFFFECh` равносильна `add esp,-20` — и здесь все верно (пять переменных $20 = 4 \times 5$).
- Далее идут еще более интересные моменты — вызов процедуры `proc2`, в листинге 3.40 это команда `call sub_4038DC`. Обращаю ваше внимание на то, что параметр в процедуру опять передается через регистр `EAX`. Но что

означает команда push ebp? Это что, еще один параметр? Но в программе его не было. Да и соглашению __fascall это не соответствует. Вспомним теперь, что вызываемая процедура proc2 является вложенной, а эта процедура должна иметь доступ к локальным переменным процедуры proc1. Вот регистр ebp и передается "тайно" в proc2, чтобы там через это значение и был доступ к локальным переменным proc1. Замечу также, что команда pop ecx, которая идет за вызовом процедуры, — это просто освобождение стека от "нелегального" параметра.

В листинге 3.41 представлен дизассемблированный код процедуры proc2 (см. листинг 3.38).

Листинг 3.41

```
CODE:004038DC  sub_4038DC  proc near   ; CODE XREF: sub_403938+29?p
CODE:004038DC    var_8  = dword ptr -8
CODE:004038DC    var_4  = dword ptr -4
CODE:004038DC    arg_0  = dword ptr  8
CODE:004038DC      push    ebp
CODE:004038DD      mov     ebp, esp
CODE:004038DF      add     esp, 0FFFFFFF8h
CODE:004038E2      mov     [ebp+var_4], eax
CODE:004038E5      mov     [ebp+var_8], 1Eh
CODE:004038EC      mov     eax, ds:off_4040A4
CODE:004038F1      mov     edx, [ebp+var_4]
CODE:004038F4      call    sub_402B78
CODE:004038F9      mov     edx, [ebp+arg_0]
CODE:004038FC      mov     edx, [edx-4]
CODE:004038FF      call    sub_402B78
CODE:00403904      mov     edx, [ebp+var_8]
CODE:00403907      call    sub_402B78
CODE:0040390C      mov     edx, [ebp+arg_0]
CODE:0040390F      mov     edx, [edx-8]
CODE:00403912      call    sub_402B78
CODE:00403917      mov     edx, [ebp+arg_0]
CODE:0040391A      mov     edx, [edx-0Ch]
CODE:0040391D      call    sub_402B78
CODE:00403922      mov     edx, [ebp+arg_0]
CODE:00403925      mov     edx, [edx-10h]
CODE:00403928      call    sub_402B78
```

```
CODE:0040392D         call    sub_402BA8
CODE:00403932         pop     ecx
CODE:00403933         pop     ecx
CODE:00403934         pop     ebp
CODE:00403935         retn
CODE:00403935   sub_4038DC    endp
```

Комментарий к листингу 3.41

- Сразу бросается в глаза множество вызовов процедур. Но мы-то знаем, что в исходном тексте (см. листинг 3.38) имеется только функция `writeln`. Однако на поверку `writeln` — вовсе не функция, а оператор. Компилятор преобразует этот оператор в вызовы двух процедур. Одна процедура (`sub_402B78`) осуществляет формирование некоторый результирующей строки, которая и будет напечатана. Процедура вызывается столько раз, сколько параметров в операторе `writeln`. После формирования результирующей строки вызывается процедура `sub_402BA8`, которая и печатает строку на консоль.
- Обратим внимание на команду `add esp,0FFFFFFF8h`. Резервируется память для двух стековых переменных. В переменную `var_4` помещается переданный в процедуру параметр. Переменная `var_8` — это локальная переменная, которой присваивается значение 30 (`1Eh`).
- Кроме двух локальных переменных в процедуре имеется параметр `arg_0`, который является не чем иным, как переданным в процедуру значением `EBP` из процедуры `proc1`, с помощью которого можно получить доступ к локальным переменным `proc1`.
- Если посмотреть на исходный текст программы, то мы увидим, что в процедуре `proc2` печатаются: `a1` — значение, переданное из `proc1` в качестве параметра, `c` — значение локальной переменной `proc1`. Кроме этого, печатаются значения четырех переменных, которые определены в `proc1`.
- Для получения значения переменных, определенных в `proc1`, используется разъясненный уже нами параметр `arg_0`. Посмотрите, как, например, извлекается значение переменной `b`:

  ```
  mov edx,[ebp+arg_0]
  mov edx,[edx-4]
  ```

  Опять же используется регистровый способ передачи параметров. В регистр же `EAX` помещается некий параметр `ds:off_4040A4`, значение которого нам неизвестно, но который, как можно догадаться, необходим для работы процедуры `sub_402B78`.

Идентификация процедур и функций (обобщение)

Идентифицировать процедуру — это значит определить, во-первых, адреса ее начала и конца, а во-вторых, количество и тип (или хотя бы размер) передаваемых параметров, а также используемых стековых переменных и, наконец, тип возвращаемого значения. Посмотрим, какие возможности у нас имеются.

Возможность 1. Вызов процедуры. Команда `CALL addr` явно указывает, что по адресу `addr` располагается некая процедура. Однако:

- косвенный вызов процедуры, например, `CALL [EAX]`, вызывает затруднение у дизассемблеров. Здесь приходится подключать отладчик, либо самому анализировать дизассемблированный текст. Если к тому же значение регистра EAX будет меняться в зависимости от значения некоторых других параметров, то обнаружить таким способом все вызываемые процедуры становится весьма затруднительно;
- из *разд. 1.6.1* мы хорошо знаем, что вызвать процедуру можно самыми разными способами, даже при помощи команды RET. И если мы имеем дело с программой или вставками на языке ассемблера, а автор имеет намерения нас запутать, то возможностей у него море. Однако при нестандартном вызове процедуры может присутствовать команда `ADD ESP,N` (или `SUB ESP,-N`, или одна или несколько команда POP), и это должно вас побудить к дополнительному анализу кода.

Возможность 2. Идентификация стандартного пролога функции. Обычно это три команды, следующие друг за другом:

```
PUSH EBP
MOV EBP,ESP
SUB ESP,N
```

Последняя команда может быть и другой, например, `ADD ESP,-N`, или просто одна или несколько команд PUSH. Наконец, выделения стека для локальных и временных переменных могут отсутствовать, если их просто нет или для этой цели используются регистры. Кроме того, в начале процедуры могут стоять команды сохранения регистров EBX, ESI, EDI. При оптимизации компилятор может обходиться без стандартного пролога и все стековые переменные и параметры адресовать при помощи регистра ESP. Наконец, в качестве пролога может быть использована команда `ENTER N`.

Конец функции проще определить, когда известно начало. Но иногда вам удается вначале идентифицировать именно окончание функции.

Возможность 3. Конец процедуры может быть определен, опять же, если имеется стандартный эпилог:

```
MOV ESP,EBP
POP EBP
```

Иногда вместо этого набора используется просто команда LEAVE. После данного эпилога естественно идет команда RETN. Вообще любая команда RETN (а особенно RETN *N*) должна настораживать — не есть ли это конец процедуры. Это хороший критерий, но срабатывает далеко не всегда. В частности, при наличии в функции более одного блока `__try...__except` компилятор Visual C++ может генерировать (ради оптимизации, разумеется) несколько стандартных эпилогов, так что и дизассемблер IDA Pro в такой ситуации легко ошибается.

Возможность 4. Конец процедуры найти проще, если определено начало. Чаще всего первая попавшаяся команда RETN и есть этот конец. Однако выйти из процедуры можно и в ее середине. Но тогда перед командой RETN вы без труда должны найти некий условный переход куда-то за команду RETN, например, так:

```
CMP EAX,1
JNZ L1
RETN
L1:
```

И дальнейший поиск конца процедуры можно продолжить с метки L1. Если процедура должна что-то возвращать по окончанию работы, т. е. является функцией, то в конце ее мы обязательно найдем команду, которая определяет значение регистра EAX: XOR EAX,EAX (возвращает false), MOV EAX,1 (возвращает true), какие-либо команды, изменяющие значение EAX (MOV, ADD, SUB и т. д.). Если тип, который возвращает функция, составляет 8 байтов, то данное возвращается в паре регистров EDX:EAX. Наконец тип double возвращается в регистре арифметического сопроцессора ST(0).

Замечание

Если возвращаемый тип является, например, структурой, то возвращается не структура, а указатель на нее в регистре EAX. При этом сама структура создается в вызывающей функции, а при вызове функции, имеющей тип "структура", через регистр EAX передается указатель на нее. Таким образом, функция будет работать с уже созданной структурой, а потом возвратит указатель на нее же.

Возможность 5. Большинство процедур и функций имеют либо переменные, определенные в стеке, либо параметры, передаваемые, опять же, через стек. Это важный признак, потому что тогда вам обязательно встретятся команды с адресацией через регистры EBP или ESP. Внимательно просматривая код над и под найденной командой, можно определить начало процедуры.

Возможность 6. При стандартной адресации стековых переменных стандартным образом (т. е. через регистр EBP) не представляет особого труда определить выделяемый для них объем стека (SUB ESP,*N* или другая подобная

команда). Что касается передаваемых параметров, то здесь проблема несколько сложнее, т. к. мы не знаем, сколько памяти на них было выделено. Проще всего проблему можно решить, найдя вызов данной процедуры, поскольку все параметры отправляются в стек обычно просто командами PUSH или другим очевидным способом (см., например, листинг 3.10 и комментарий к нему) Если место, откуда была вызвана наша процедура, не известно, то придется анализировать ее текст. Для начала следует найти максимальное смещение при адресации относительно значения EBP в сторону старших адресов. Поскольку после параметров в стек положили еще адрес возврата и старое значение EBP, то первый (с минимальным адресом) параметр будет находиться по адресу [EBP+8] (см. рис. 3.2). Таким образом, если максимальное смещение при использовании адресации [EBP+N] равно max_off, то количество байтов, которое было выделено для параметров, составит max_off-4, а ориентировочное количество параметров из предположения, что все они 32-битные и имеют простой тип (не массивы или структуры), составит (max_off-4)/4.

После теоретических выкладок приведем конкретный пример. Рассмотрим следующую программу, написанную на C++ (листинг 3.42).

Листинг 3.42

```
#include <stdio.h>
#include <windows.h>
double myfunc(double, __int64, int, BYTE);
void main()
{
  double ff=10.45;
  __int64 ii=1000;
  int jj=200;
  BYTE bb=50;
  double ss=myfunc(ff,ii,jj,bb);
  printf("%f\n",ff);
};
double myfunc(double f, __int64 i, int j, BYTE b)
{
  double s;
  s=f+i+j+b;
  printf("%f\n",s);
  return s;
};
```

В листинге 3.43 содержится дизассемблированная функция `main` из листинга 3.42.

Листинг 3.43

```
.text:00401000  _main    proc near    ; CODE XREF: start+16E?p
.text:00401000    var_40  = qword ptr -40h
.text:00401000    var_30  = qword ptr -30h
.text:00401000    var_28  = qword ptr -28h
.text:00401000    var_1C  = dword ptr -1Ch
.text:00401000    var_18  = qword ptr -18h
.text:00401000    var_10  = dword ptr -10h
.text:00401000    var_C  = dword ptr -0Ch
.text:00401000    var_1  = byte ptr -1
.text:00401000      push    ebp
.text:00401001      mov     ebp, esp
.text:00401003      sub     esp, 28h
.text:00401006      fld     ds:dbl_408108
.text:0040100C      fstp    [ebp+var_28]
.text:0040100F      mov     [ebp+var_10], 3E8h
.text:00401016      mov     [ebp+var_C], 0
.text:0040101D      mov     [ebp+var_1C], 0C8h
.text:00401024      mov     [ebp+var_1], 32h
.text:00401028      mov     al, [ebp+var_1]
.text:0040102B      push    eax
.text:0040102C      mov     ecx, [ebp+var_1C]
.text:0040102F      push    ecx
.text:00401030      mov     edx, [ebp+var_C]
.text:00401033      push    edx
.text:00401034      mov     eax, [ebp+var_10]
.text:00401037      push    eax
.text:00401038      fld     [ebp+var_28]
.text:0040103B      sub     esp, 8
.text:0040103E      fstp    [esp+40h+var_40]
.text:00401041      call    sub_401070
.text:00401046      add     esp, 18h
.text:00401049      fstp    [ebp+var_18]
.text:0040104C      fld     [ebp+var_28]
```

```
.text:0040104F      sub     esp, 8
.text:00401052      fstp    [esp+30h+var_30]
.text:00401055      push    offset unk_4080FC
.text:0040105A      call    _printf
.text:0040105F      add     esp, 0Ch
.text:00401062      xor     eax, eax
.text:00401064      mov     esp, ebp
.text:00401066      pop     ebp
.text:00401067      retn
.text:00401067 _main    endp
```

Комментарий к листингу 3.43

- Проведем идентификацию четырех локальных переменных, определенных в функции main. Отбросим вначале имена var_30 и var_40 — это обозначения IDA Pro, и переменными они не являются. Для локальных переменных отводится 40 байтов. Это слишком много для пяти переменных. Но давайте все по порядку. Итак, var_28 очевидно представляет переменную ff, имеющую тип double. Здесь все ясно, загрузка начального значения осуществляется командами fld/fstp из константы dbl_408108. Команды

  ```
  mov [ebp+var_10],3E8h
  mov [ebp+var_C],0
  ```

  очевидно, загружают в переменную ii значение 1000 (3E8h). Дизассемблер не понимает, что это одна 64-битная переменная, и считает их двумя разными переменными. var_1C обозначает переменную jj.

  Далее — переменная var_1, она однобайтовая и обозначает переменную bb. Обратите внимание: несмотря на то, что переменная однобайтовая, она, по сути, занимает 4 байта, а далее имеются еще четыре свободных байта, и только за ними начинается переменная var_C. Это так компилятор выровнял данные по границе 8 байтов. Уже это может насторожить и вызвать предположение, что далее идут не две переменные по 4 байта, а одна в 8 байтов.

  У нас осталась еще переменная ss. Заметьте, что после вызова функции myfunc, которая имеет тип double, стоит команда fstp [ebp+var_18], т. е. в переменную var_18 загружается значение из регистра ST(0). Но тип double как раз и возвращается в регистре ST(0), так что var_18 и есть переменная ss.

  Итак, все нормально, все переменные найдены, а лишнее резервирование оказалось связанным с выравниванием данных.

- Вызывает интерес последовательность команд

  ```
  mov al,[ebp+var_1]
  push eax
  ```

  "Что же здесь не ясного?" — скажете вы. Ведь переменная-то однобайтовая, а в стек следует отправлять четырехбайтовую величину. Все так, но старшие байты регистра `EAX` остались неочищенными. И далее все двойное слово отправляется в стек, как параметр. Очевидно, такое возможно при одном условии, если в функции будет строго учитываться, что параметр — однобайтовый. Кстати, обратите внимание на порядок, в котором параметры отправляются в стек. Параметры отправляются справа налево. При этом освобождает стек вызывающая функция. Это и есть соглашение __cdecl (см. табл. 3.2). Далее в стек отправляются все остальные переменные. Переменная `ii` (`var_10`, `var_c`) отправляется в стек как две независимые 4-байтовые переменные. Ну, а переменная `ff`, как и должно быть, отправляется в стек посредством команды `fstp`. Далее идет вызов функции `printf`, и здесь для нас уже нет ничего необычного.

  Но вот на что я бы обратил ваше внимание. Первым параметром этой функции идет форматная строка, в которой указывается спецификация остальных параметров функции. Эта спецификация часто помогает нам определить тип и размер переменной, тем более, что функций, подобных `printf`, оперирующих форматной строкой, в библиотеке C++ несколько.

В листинге 3.44 содержится дизассемблированный код функции `myfunc`.

Листинг 3.44

```
\
.text:00401070    sub_401070  proc near  ; CODE XREF: _main+41?p
.text:00401070    var_14  = qword ptr -14h
.text:00401070    var_C  = dword ptr -0Ch
.text:00401070    var_8  = qword ptr -8
.text:00401070    arg_0  = qword ptr  8
.text:00401070    arg_8  = qword ptr  10h
.text:00401070    arg_10  = dword ptr  18h
.text:00401070    arg_14  = byte ptr  1Ch
.text:00401070      push    ebp
.text:00401071      mov     ebp, esp
.text:00401073      sub     esp, 0Ch
.text:00401076      fild    [ebp+arg_8]
.text:00401079      fadd    [ebp+arg_0]
.text:0040107C      fiadd   [ebp+arg_10]
```

```
.text:0040107F          movzx   eax, [ebp+arg_14]
.text:00401083          mov     [ebp+var_C], eax
.text:00401086          fild    [ebp+var_C]
.text:00401089          faddp   st(1), st
.text:0040108B          fst     [ebp+var_8]
.text:0040108E          sub     esp, 8
.text:00401091          fstp    [esp+14h+var_14]
.text:00401094          push    offset byte_408100
.text:00401099          call    _printf
.text:0040109E          add     esp, 0Ch
.text:004010A1          fld     [ebp+var_8]
.text:004010A4          mov     esp, ebp
.text:004010A6          pop     ebp
.text:004010A7          retn
.text:004010A7  sub_401070 endp
```

Комментарий к листингу 3.44

- Начнем с разбора стековых переменных. Их всего две (var_14 не в счет): var_8 и var_c. Переменная var_8 занимает 8 байтов, и это наводит на мысль, что это есть не что иное, как переменная s. Это предположение подтвердится нами в дальнейшем. Других переменных в функции myfunc не объявлялось и, следовательно, четырехбайтовая переменная var_c — это просто временная переменная.
- Теперь обратимся к параметрам. Удивительно, но их всего четыре. То есть мы-то знаем, что их четыре, но в функции main IDA Pro посчитал, что имеется пять переменных, которые затем используются в качестве параметров. Здесь все просто. В функции main у дизассемблера не было веских оснований считать, что var_10 и var_c — это одна переменная, а вот в функции myfunc у дизассемблера есть все основания считать, что arg_8 — это один 8-байтный параметр, т. е. число __int64 (см. команду fild).
- Разберем алгоритм вычисления выражения f + i + j + b. Итак, команда fild загружает длинное целое число (т. е. число i) в вершину стека сопроцессора, т. е. ST(0). Следующая команда fadd складывает это число с вещественным числом, т. е. f. Результат при этом помещается в ST(0) и трактуется как вещественный. Следующая команда fiadd складывает вещественное число, хранящееся в ST(0) с целым 32-битным числом j. Результат опять помещается в ST(0). Далее команда movzx eax,[ebp+arg_14] помещает байт в регистр EAX и очищает старшие байты регистра. В комментарии к листингу 3.43 это уже обсуждалось. Байт от-

правляется в стек в составе двойного слова, при этом вызывающая сторона не очищает старшие байты, а вот вызываемая процедура это делает. Иначе ошибка неминуема. Далее используется временная переменная `var_c`, куда помещается число `b` (`mov [ebp+var_C],eax`), которая затем загружается в регистр `ST(0)`, при этом старое значение `ST(0)` перемещается в `ST(1)`. Наконец, команда `faddp st(1),st` и результат вычисления `f + i + j + b` помещается в `var_8` (переменная `s`) — команда `fst [ebp+var_8]`. При этом стек не выталкивается, и результат по-прежнему хранится в `ST(0)`. Так что последовательность команд

```
sub esp,8
fstp [esp+14h+var_14]
```

помещает этот результат в стек для вывода при помощи функции `printf`. Наконец, последний штрих: `fld [ebp+var_8]` — это возвращаемое функцией значение. Конечно, здесь компилятор сделал промашку. Не нужно было использовать команду `fstp`, тогда и последняя команда не понадобилась бы.

Переполнение буфера

Переполнение буфера (buffer overflows) — это один из методов корректировки программного обеспечения во время его выполнения. Взломщик путем умелого ввода данных осуществляет передачу управления внедренному в программу коду. Мы будем рассматривать только одну разновидность — *переполнение стека* (stack overflows). Переполнение стека наиболее ярко себя проявляет в программах, написанных на языке C++, и заключается в проникновении в исполняемый код через стек программы.

Прием "переполнения стека" осуществляется, в основном, для проведения удаленных атак, поскольку если вы пытаетесь проникнуть в программу, которая у вас есть на компьютере, то в вашем распоряжении находятся куда более мощные средства. С другой стороны, взлом некоторой системы на удаленном компьютере предполагает проведение предварительных исследований нужной для вас программы. Именно по этой причине я и поместил данный материал в мою книгу, хотя он более относится к рассмотрению удаленных атак в сети.

Суть проблемы

При работе с внешними устройствами программа выделяет буферы для хранения полученных и отправляемых данных. При получении данных они полностью или частично заполняют выделенные буферы. Программа должна заботиться о том, чтобы полученные данные не выходили за границы буфера. В противном случае могут быть повреждены другие данные или даже программный код. Так или иначе, выход загружаемых данных за выделен-

ные для них границы может привести к частичной или полной неработоспособности программы. Особенностью таких ошибок является то, что они чрезвычайно трудно выявляются. В некоторых случаях у программиста может создаться впечатление, что ошибки возникают случайно и не связаны с какими-либо действиями. Типичным примером переполнения является выход за границы массива.

Многие современные компиляторы обладают возможностью генерировать код, способный автоматически проверять, не было ли выхода за границы выделенных буферов. Например, в компиляторе Microsoft C++ существует опция `/GS`, при использовании которой генерируется дополнительный код, осуществляющий проверку операций на предмет выхода за границы буферов. Правда, проверке подвергаются лишь буферы, определенные в стеке, да и то весьма поверхностно, абсолютно не гарантируя попадания данных из одного буфера в другой. Суть идеи проверки на переполнение стека заключается в том, чтобы по краям выделяемых буферов расположить заранее известные байты. После любых операций записи в буфер должна вызываться процедура проверки этих байтов. Искажение этих байтов должно означать, что произошел выход за пределы буфера, т. е. его переполнение. Разумеется, данный подход требует, во-первых, дополнительной памяти, а во-вторых, естественно, замедляет работу программы.

Возникает законный вопрос: как переполнение буфера может быть использовано теми, кто пытается взломать программу или систему? Здесь существует несколько путей проникновения.

- Если буфер располагается в стеке, то самым очевидным механизмом воздействия извне будет искажение адреса возврата из функции. Адрес возврата может быть искажен таким образом, что переход будет осуществлен на другую функцию или же по адресу, который расположен здесь же в стеке, куда вместо обычных данных помещен вредоносный исполняемый код. Именно этот способ проникновения мы будем рассматривать далее.
- В процессе переполнения могут быть искажены какие-либо указатели (указатели на функции, таблица переходов и т. п.) так, что они будут указывать на совсем иной код, который станет выполняться согласно планам взломщика.
- В процессе переполнения будут искажены адреса данных так, что следующий ввод будет осуществляться совсем в иное место программы, что опять позволит взломщику внедрить в исполняемую программу свой код.

Из сказанного следует, что взлом программного обеспечения по методу переполнения буфера осуществляется в два приема: внедрение вредоносной программы и передача управления на вредоносную программу. Теоретически первое может и отсутствовать, если нужная для вас процедура является частью исследуемой программы. В этом случае вам необходимо лишь в нужный момент умело передать управление на эту процедуру.

Почему мы рассматриваем именно переполнение стека. Все дело в том, что Windows защищает код исполняемой программы от записи туда (тема модификации кода во время его выполнения была подробно нами освещена в *разд. 1.6.2*), а область данных — от исполнения в ней некоторого кода. И только в стеке можно писать и исполнять данные. Причем это свойство стека на поверку оказывается универсальным для большинства операционных систем. Таким образом, мы можем заполнить стековый буфер исполняемым кодом, а затем заставить процессор исполнить его.

Пример

В качестве примера рассмотрим следующую простую программу (листинг 3.45). Функция `getpassword` осуществляет проверку правильности пароля и в зависимости от этого возвращает `false` или `true`.

Листинг 3.45

```
#include <stdio.h>
#include <string.h>
int getpassword(char *);
char * passw="privet";
int main()
{
  printf("Input password:\n");
  if(!getpassword(passw))printf("You are registered!\n");
  else printf("You are wrong!\n");
  return 0;
};

int getpassword(char * ss)
{
  char s[13];
  gets(s);
  if(!strcmp(s,ss))return 0;
  else return 1;
}
```

На рис. 3.3 представлена схема стека программы из листинга 3.45. Я взял стандартную схему с прологом и эпилогом. Как видите, общая структура стека распадается на структуру стека функции `main` и структуру стека

функции getpassword. Обратите внимание, что адреса уменьшаются снизу вверх. Локальная переменная на рисунке — это, разумеется, переменная s. И хотя размер переменной задан нами в 13 байтов, компилятор выравнивает его по границе в 4 байта, так что наш буфер оказывается равным 16 байтам. Ввод данных в переменную s осуществляется от меньших адресов к большим. Таким образом, данные как бы "наползают" на все то, что расположено ниже в стеке. Первым опасности подвергается значение EBP. А вот следом идет адрес возврата из функции getpassword. Это и есть наша вожделенная цель. Если данные выйдут за пределы буфера и изменят значение адреса возврата, то возврат будет осуществлен в совсем иное место. "Что это за место?" — спросите вы. И это самый интересный вопрос. Мы вполне можем поместить в стек адрес какой-нибудь имеющейся в программе функции, так что программа будет выполняться по совершенно непредвиденному программистом пути.

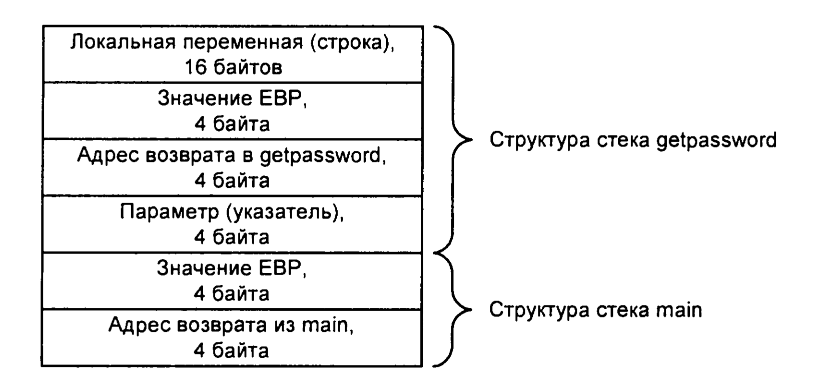


Рис. 3.3. Структура стека. Адреса уменьшаются снизу вверх

Вот это мы сейчас и постараемся сделать, т. е. изменить адрес возврата из функции getpassword. Для начала надо внимательно изучить дизассемблированный исполняемый код программы. В листингах 3.46 и 3.47 представлены дизассемблированные тексты функций main и getpassword соответственно.

Листинг 3.46

```
.text:00401000  _main    proc near      ; CODE XREF: start+16E?p
.text:00401000       push    ebp
.text:00401001       mov     ebp, esp
```

```
.text:00401003      push    offset aInputPassword
.text:00401008      call    _printf
.text:0040100D      add     esp, 4
.text:00401010      mov     eax, dword_409040
.text:00401015      push    eax
.text:00401016      call    sub_401050
.text:0040101B      add     esp, 4
.text:0040101E      test    eax, eax
.text:00401020      jnz     short loc_401031
.text:00401022      push    offset aYouAreRegister
.text:00401027      call    _printf
.text:0040102C      add     esp, 4
.text:0040102F      jmp     short loc_40103E
.text:00401031  loc_401031:            ; CODE XREF: _main+20?j
.text:00401031      push    offset aYouAreWrong
.text:00401036      call    _printf
.text:0040103B      add     esp, 4
.text:0040103E  loc_40103E:            ; CODE XREF: _main+2F?j
.text:0040103E      xor     eax, eax
.text:00401040      pop     ebp
.text:00401041      retn
.text:00401041  _main    endp
```

Комментарий к листингу 3.46

- Начну с вызова функции `sub_401050`, которая является не чем иным, как обозначением функции `getpassword`. Последовательность команд

  ```
  mov eax, dword_409040
  push eax
  ```

  это отправка в стек указателя на строку, содержащую пароль-образец. Глобальная переменная `dword_409040` содержит адрес этой строки, т. е. является переменной-указателем. Таким образом, в стек вначале отправляется параметр, а далее команда `call` помещает туда адрес возврата, т. е. `0040101Bh`.
- Далее обратим внимание на команду `test eax,eax` и следующую за ней команду условного перехода `jnz short loc_401031`. Конечно, это обычная условная конструкция и команда `test` соответствует оператору `if`. Дальнейшая структура `main` вполне нам знакома.

Листинг 3.47

```
.text:00401050  sub_401050  proc near  ; CODE XREF: _main+16?p
.text:00401050    var_10  = byte ptr -10h
.text:00401050    arg_0  = dword ptr  8
.text:00401050      push    ebp
.text:00401051      mov     ebp, esp
.text:00401053      sub     esp, 10h
.text:00401056      lea     eax, [ebp+var_10]
.text:00401059      push    eax
.text:0040105A      call    _gets
.text:0040105F      add     esp, 4
.text:00401062      mov     ecx, [ebp+arg_0]
.text:00401065      push    ecx             ; char *
.text:00401066      lea     edx, [ebp+var_10]
.text:00401069      push    edx             ; char *
.text:0040106A      call    _strcmp
.text:0040106F      add     esp, 8
.text:00401072      test    eax, eax
.text:00401074      jnz     short loc_40107A
.text:00401076      xor     eax, eax
.text:00401078      jmp     short loc_40107F
.text:0040107A  loc_40107A:    ; CODE XREF: sub_401050+24?j
.text:0040107A      mov     eax, 1
.text:0040107F  loc_40107F:    ; CODE XREF: sub_401050+28?j
.text:0040107F      mov     esp, ebp
.text:00401081      pop     ebp
.text:00401082      retn
.text:00401082  sub_401050  endp
```

Комментарий к листингу 3.47

- Прежде всего, отмечу тот факт, что на локальные переменные в функции отводится 16 байтов. Я уже говорил об этом. Это связано с выравниванием всех данных в стеке по 4-байтовой границе. Так что, когда мы станем "переполнять" стек, то будем иметь в виду реальный размер этого буфера.
- Последовательность команд

```
lea eax,[ebp+var_10]
push eax
```

просто помещает в стек адрес нашего буфера, куда, как предполагается, следует поместить вводимый пароль. Вот это и есть ключевой момент. Как вы понимаете (см. рис. 3.3), за этим буфером идет содержимое `EBP`, а далее желанный адрес возврата.

- После вызова библиотечной функции `gets` вызывается функция сравнения строк `strcmp`. Функция получает в стеке адреса двух строк. После выполнения функции — обычная условная конструкция (`test`, а потом `jnz`). Замечу, кстати, что функция `strcmp`, как и многие другие строковые функции, контролирует длину строк только по конечному значению 0, а, следовательно, переполнение буфера абсолютно не может затронуть их работу.

Итак, после разбора листингов можно приступать к действию. Что, собственно, мы хотим? Давайте попытаемся изменить адрес возврата, чтобы переход по `retn` из функции `getpassword` происходил на команду `printf` в начале функции `main`. Из листинга 3.46 следует, что адрес перехода равен `00401003`. Вспомним, что в памяти число записывается по принципу "старшему байту — старший адрес", и получим, что в буфер следует отправить последовательность байтов `03 10 40 00`. Но сперва следует заполнить 16-байтовый буфер, затем еще 4 байта, где лежит значение `EBP`. Поскольку в командной строке сложно ввести символы с кодами `10h`, `03h`, то лучше использовать следующий прием. Подготовим текстовый файл с нужной строкой, а затем воспользуемся перенаправлением ввода. Итак, если наша программа имеет имя prog1.exe, а текстовый файл — имя pasw.txt, то следует выполнить такую команду:

```
prog1 < pasw.txt
```

Для ввода же символов с кодами, меньшими, чем 32, можно использовать программу hiew.exe *(см. разд. 2.1.3)*.

Итак, вот наша строка:

```
qqqqqqqqqqqqqqqqqqqq??@
```

Ровно 20 байтов (16 байтов — буфер строки и 4 байта — содержимое `EBP`) мы заполняем произвольными символами, например, символом `q`. Далее идут символы с кодами `03h`, `10h`, `40h`. "А где же символ с кодом 0?" — спросите вы. А зачем он нам? Ведь в адресе, который мы меняем, он и так есть, и стоит он на том месте, где надо.

А теперь приготовились! Выполняем команду `prog1 < pasw.txt`. Вот что мы имеем:

```
Input password:
Input password:
You are wrong!
```

Задача решена. Действительно, после выполнения функции `getpassword` происходит переход по адресу, который мы и указали в буфере. Правда, после вывода этих строк появляется окно (рис. 3.4), предупреждающее, что возникла критическая ситуация (исключение). Но это должно быть понятно. Ведь при втором проходе в буфер попадают совсем другие данные, а стек уже испорчен, и приложение не может правильно закончить свою работу.

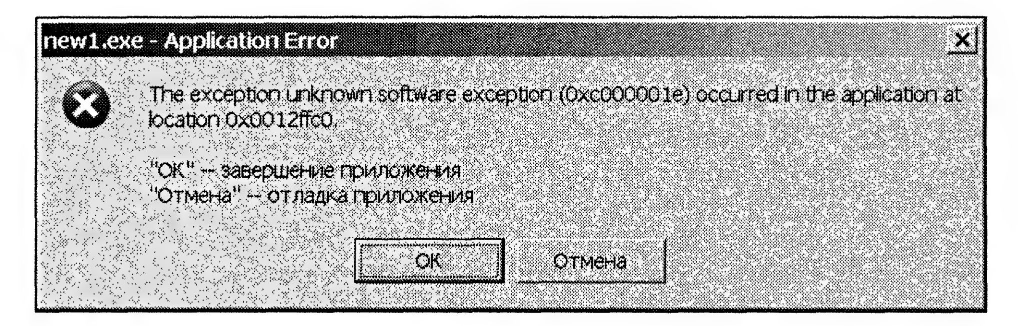


Рис. 3.4. Сообщение операционной системы Windows 2003

Замечание

Конечно, у читателя возникает следующий вопрос: получается так, что мы сильно ограничены тем, какие байты отправляем в буфер? Вам не удастся, например, отправить в буфер символ с кодом 26 или 0. Замечание справедливое, но:

- во-первых, мы ведь рассматриваем частный случай — ввод с помощью консольной функции `gets`. В общем случае буфер может быть предназначен и для произвольной двоичной информации, и тогда мы будем абсолютно свободны в выборе информации, которую станем отправлять на ввод;
- во-вторых, информация, которая будет передаваться в буфер, может быть закодирована так, что в ней не окажется "опасных" кодов. Но об этом мы будем говорить немного ниже.

Теперь вспомним, что в стеке можно исполнять программу. А что, если в стек поместить программный код и передать управление на этот код. Не плохо, да? Это в нашей программе буфер составляет всего 16 байтов, а представьте себе, что мы имеем дело с буфером размером 16 Кбайт. Да в такой буфер можно поместить программу, которая сделает все, что мы захотим, причем от имени программы, которая, возможно, имеет в системе очень высокие права. Вот вам лазейка, которой пользуются взломщики уже более десяти лет.

Ну, да ладно, а что, собственно мы в данном случае можем поместить? Да хотя бы следующий код:

```
MOV EAX,0
RETN
```

Однако я ошибся, ведь команда RETN уже выполнена для перехода на данный фрагмент, так что надо вместо RETN поставить какую-нибудь разновидность команды JMP.

Если это удастся, то наша программа окажется взломанной, поэтому она всегда на нашу строку будет реагировать, что пароль верен.

Взглянув на листинг 3.46, легко сообразить, что адрес перехода должен быть 0040101B. Тогда состояние стека будет в полном порядке и не возникнет никаких критических ошибок.

К нашему огорчению последовательность

```
MOV EAX,0
JMP 0040101B
```

не годится для передачи ее в качестве строки, поскольку:

- первая команда содержит нули; код команды MOV EAX,0 равен B8 00000000:
- во второй команде адрес перехода отсчитывается относительно команды, следующей за командой JMP. Если адрес начала стека будет изменен, то наш фрагмент перестанет работать корректно.

По этой причине две наших старых команды превращаются в 6 следующих команд:

```
XOR EAX,EAX      ;33 C0
XOR ECX,ECX      ;33 C9
MOV CL, 40H      ;B1 40
SHL ECX,10H      ;C1 E1 10
MOV CX, 101BH    ;66 B9 1B 10
JMP ECX          ;FF E1
```

Справа от команд я указал код этих команд. Для получения правильного кода всех команд лучше воспользоваться каким-нибудь отладчиком, например, OllyDbg. Итак, мы потратили 15 байтов из 20 возможных (16 байтов, отведенных для строки, и 4 байта для хранения EBP). Оставшиеся 5 байтов могут содержать любую информацию. Идущий же далее адрес возврата должен содержать адрес начала буфера. Адрес буфера можно найти из того же отладчика, и он оказывается равным 0012FEC8. Нам, таким образом, к строке из 20 байтов следует добавить еще три байта: C8 FE 12.

Вот содержание файла pasw.txt:

```
3L3r-@+c_f¦__ cqqqqq+¦_
```

Совет

Во избежание ошибок заполняйте текстовый файл при помощи программы hiew.exe, а не текстовым редактором. Попытка переноса некоторых символов через буфер обмена неизбежно приведет к искажению кода, при этом сам символ визуально может остаться тем же.

Итак, все готово, и выполняем команду

```
progl < pasw.txt
```

И, о чудо!

```
You are registered!
```

Таким образом нам удалось внедрить в программу свой код, заставить программу нас зарегистрировать.

Осталось обсудить еще один вопрос. Как мы видели, есть проблема с набором символов, которые можно отправлять программе в качестве строки. Разумеется, не всегда ввод осуществляется при помощи консольной процедуры, избирательно относящейся к некоторым символам. Если мы имеем дело с таким вводом, то, следовательно, и проблемы такой нет.

Однако все-таки вернемся к случаю консольного ввода. Так каково же решение проблемы? Ответ прост: надо закодировать последовательность байтов, чтобы в них отсутствовали соответствующие коды. Я не буду разбирать различные способы кодирования. Мне нравится следующий подход, который, однако, может потребовать дополнительной памяти. Суть его заключается в следующем. Нужно закодировать все "непроходные" байты (например, командой `XOR`). Перед каждым таким байтом должен идти байт, определяющий, что следующий за ним байт закодирован. Естественно использовать для этой цели команду `NOP`, имеющую код `90H`. Разумеется, весь фрагмент должен начинаться с декодировщика остальной части кода. Декодирование сводится к удалению байтов `NOP` и декодировке следующих за ним байтов. Поскольку байтов, не воспринимаемых (или нестандартно воспринимаемых) функциями консольного ввода, не так уж и много, то количество команд `NOP` не должно быть уж слишком большим.

На этом я закончу рассказ о переполнении буфера.

3.2.2. Условные конструкции и операторы выбора

Программируя на языках высокого уровня, таких как C или Pascal, мы уже привыкли к использованию полных условных конструкций (`if...else`) и

логических связок (and, or). Но ведь были времена, когда этого не было. Возьмите, например, язык FORTRAN или ранний Basic. Вот тогда на помощь приходил оператор безусловных переходов (goto), который так не любят ревнители высокого стиля в программировании. Однако машинный язык весь построен на операторах переходов условных или безусловных, без них невозможно обойтись, если хотите проверить какое-либо условие.

Простые конструкции

Рассмотрим простую условную конструкцию (листинг 3.48).

Листинг 3.48

```
#include <stdio.h>
void main()
{
  int a,b;
  scanf("%d",&a);
  scanf("%d",&b);
  if(a>=b)
    printf("a>=b\n");
  else
    printf("a<b\n");
}
```

После компилирования в Microsoft Visual Studio и загрузки исполняемого модуля в IDA Pro получим листинг 3.49.

Листинг 3.49

```
.text:00401000  _main    proc near  ; CODE XREF: start+16E?p
.text:00401000    var_8  = dword ptr -8
.text:00401000    var_4  = dword ptr -4
.text:00401000      push    ebp
.text:00401001      mov     ebp, esp
.text:00401003      sub     esp, 8
.text:00401006      lea     eax, [ebp+var_4]
.text:00401009      push    eax
.text:0040100A      push    offset unk_4080FC
.text:0040100F      call    _scanf
```

```
.text:00401014      add     esp, 8
.text:00401017      lea     ecx, [ebp+var_8]
.text:0040101A      push    ecx
.text:0040101B      push    offset unk_408100
.text:00401020      call    _scanf
.text:00401025      add     esp, 8
.text:00401028      mov     edx, [ebp+var_4]
.text:0040102B      cmp     edx, [ebp+var_8]
.text:0040102E      jl      short loc_40103F
.text:00401030      push    offset aAB       ; "a>=b\n"
.text:00401035      call    _printf
.text:0040103A      add     esp, 4
.text:0040103D      jmp     short loc_40104C
.text:0040103F  loc_40103F:      ; CODE XREF: _main+2E?j
.text:0040103F      push    offset aAB_0     ; "a<b\n"
.text:00401044      call    _printf
.text:00401049      add     esp, 4
.text:0040104C  loc_40104C:      ; CODE XREF: _main+3D?j
.text:0040104C      xor     eax, eax
.text:0040104E      mov     esp, ebp
.text:00401050      pop     ebp
.text:00401051      retn
.text:00401051  _main    endp
```

Комментарий к листингу 3.49

- Обратите внимание, как вызывается функция `scanf`. Поскольку своим аргументом она требует указатель на переменную, поэтому

  ```
  lea eax,[ebp+var_4]
  push eax
  ```

  отправляем указатель на переменную `var_4` в стек. Аналогично компилятор Microsoft поступает и с переменной `var_8`.

- Второй важный момент — как организуется полная условная конструкция в исполняемом коде. Схематично это можно представить так:

  ```
  jl  l1
  //a>=b
  ...
  jmp l2
  ```

```
  l1:
     //a<b
     ...
  l2:
```

Как видим, для реализации полной условной конструкции требуется один условный и один безусловный переход. Обратите внимание, что команда условного перехода соответствует условию, являющемуся отрицанием условия в оригинальном тексте программы. Если уберем в программе ветку `else` (неполная условная конструкция), то в исполняемом коде просто исчезнет команда безусловного перехода (`jmp l2`). Наконец, если изменить условие с `a>=b` на `a>b`, то в исполняемом коде команда `jl` изменится на `jle` (меньше или равно). В том случае, если в исходной программе используется условие "меньше или равно", например, `a=<b`, в исполняемом коде применяется команда `jg` или `jge` (для условия "меньше"). Тот факт, что вместо прямого условия в исполняемом коде проверяется его отрицание, совсем не является аксиомой. Возможен и другой подход:

```
   jge  l1
   //a<b
   ...
   jmp l2
l1:
   //a>=b
   ...
l2:
```

Как видим, в нем нет ничего аномального. Вы вполне можете встретиться с ним при анализе кода, созданного каким-нибудь компилятором. Однако подчеркну еще раз, мы не занимаемся декомпиляцией, а стараемся понять логику исполняемого кода, и для этого совсем не обязательно точно знать, какой программный текст породил данный фрагмент.

Если переменные в программе — беззнаковые, тогда вместо `jl` (`jle`) используется `jb` (`jbe`), а вместо `jg` (`jge`) указывается `ja` (`jae`). В случае проверки на равенство (==) или неравенство (`!=`) используется, соответственно, `jnz` и `jz`.

Отмечу, что в разобранном примере для проверки условия `a>b` в исполняемом коде использовалась команда `CMP`. Это довольно очевидно. Эта команда используется для проверки и других условий: <, <=, >=, ==, `!=`. Все зависит от того, какой условный переход вы затем применяете, другими словами, какой флаг (или группу флагов) проверяете. В случае проверки равенства (или неравенства) нулю вместо команды `CMP` чаще используется команда `TEST`. Напоминаю, что во многих языках программирования значение `false`

(ложь) соответствует числовому значению 0, а значение `true` (истина) — ненулевому значению (например, 1). Интересно в этой связи рассмотреть типичную для языка C++ конструкцию:

```
if(k=(a==b)))
{
} else
{
}
```

Ясно, что переменной `k` будет присвоено одно из двух значений: 1 (если `a` равно `b`) и 0 (если `a` неравно `b`). Вот интересующий нас и прокомментированный мной фрагмент исполняемого кода, сформированный компилятором Visual C++:

```
;помещаем переменную a в регистр EAX
mov  eax, [ebp+var_4]
;вычисляем разность a-b, при этом сами переменные остаются неизменными
sub  eax, [ebp+var_8]
;смена знака, по сути, нужна для определения, 0 или нет в регистре EAX
neg  eax
;вычитание с учетом знака
;если в EAX был не 0, то вычитание даст в EAX -1,
;в противном случае в EAX будет 0
sbb  eax, eax
;если в EAX было -1, то в результате команды inc
;будет 0 (false), в противном случае будет 1 (true)
inc  eax
;значение в переменную k
mov  [ebp+var_C], eax
;переходим в соответствии с тем, что в регистре EAX
jz   short loc_401058
...
jmp  short loc_401065
loc_401058:
...
loc_401065:
...
```

Не правда ли, алгоритм получения значения переменной `k` весьма примечателен? Команда `CMP`, как видите, в данном случае не применяется. Обратите внимание, что условный переход в данном случае осуществляется в соответ-

ствии со значением, которое оказалось в регистре `EAX` после операции `INC EAX`. Но в `EAX` получается либо 0 (`false`), либо 1 (`true`).

Отдельно следует остановиться на сравнении вещественных чисел (листинг 3.50).

Листинг 3.50

```
#include <stdio.h>
void main()
{
  double a,b;
  scanf("%Lf",&a);
  scanf("%Lf",&b);
  if(a>=b)
    printf("%Lf\n",a);
  else
    printf("%Lf\n",b);
}
```

В листинге 3.50 представлена простая программа, в которой сравниваются два вещественных числа типа `double`. С точки зрения синтаксиса языка, разница между аналогичной программой с целыми переменными и данной программой минимальна и касается формата функций `scanf` и `printf`. Однако сравнение вещественных переменных должно кардинально отличаться от сравнения целых переменных на уровне исполняемого кода.

Смотрим листинг 3.51, который предоставляет нам IDA Pro.

Листинг 3.51

```
.text:00401000  _main    proc near  ; CODE XREF: start+16E?p
.text:00401000    var_18  = qword ptr -18h
.text:00401000    var_10  = qword ptr -10h
.text:00401000    var_8  = qword ptr -8
.text:00401000      push    ebp
.text:00401001      mov     ebp, esp
.text:00401003      sub     esp, 10h
.text:00401006      lea     eax, [ebp+var_8]
.text:00401009      push    eax
.text:0040100A      push    offset unk_4090FC
.text:0040100F      call    _scanf
```

```
.text:00401014      add     esp, 8
.text:00401017      lea     ecx, [ebp+var_10]
.text:0040101A      push    ecx
.text:0040101B      push    offset unk_409100
.text:00401020      call    _scanf
.text:00401025      add     esp, 8
.text:00401028      fld     [ebp+var_8]
.text:0040102B      fcomp   [ebp+var_10]
.text:0040102E      fnstsw  ax
.text:00401030      test    ah, 1
.text:00401033      jnz     short loc_40104D
.text:00401035      fld     [ebp+var_8]
.text:00401038      sub     esp, 8
.text:0040103B      fstp    [esp+18h+var_18]
.text:0040103E      push    offset aLf      ; "%Lf\n"
.text:00401043      call    _printf
.text:00401048      add     esp, 0Ch
.text:0040104B      jmp     short loc_401063
.text:0040104D  loc_40104D:     ; CODE XREF: _main+33?j
.text:0040104D      fld     [ebp+var_10]
.text:00401050      sub     esp, 8
.text:00401053      fstp    [esp+18h+var_18]
.text:00401056      push    offset aLf_0    ; "%Lf\n"
.text:0040105B      call    _printf
.text:00401060      add     esp, 0Ch
.text:00401063  loc_401063:     ; CODE XREF: _main+4B?j
.text:00401063      xor     eax, eax
.text:00401065      mov     esp, ebp
.text:00401067      pop     ebp
.text:00401068      retn
.text:00401068  _main   endp
```

Комментарий к листингу 3.51

- В выделении памяти для стековых переменных для нас уже нет ничего нового. Замечу только, что переменные имеют тип `double` и поэтому занимают 8 байтов. Аргументами функций `scanf` являются не сами переменные, а указатели на них — отсюда использование команды lea:

```
lea eax,[ebp+var_8]
push eax
```

Использование двух регистров (EAX, ECX) в качестве временных переменных для хранения указателей на переменные для дальнейшей отправки в стек, конечно, не несет в себе никакого особого смысла. Вполне можно было бы обойтись и одним регистром.

- Далее с адреса 00401028 начинается действительно интересное. Ведь сравнить надо два восьмибайтовых вещественных числа. Итак, вот последовательность команд:

```
;загрузить переменную a в ST(0)
fld    [ebp+var_8]
;сравнить содержимое ST(0) с var_10 (переменная b)
fcomp  [ebp+var_10]
;сохранить слово состояния (SW) в регистре AX
fnstsw ax
;проверить нулевой бит регистра AH
test   ah, 1
;перейти, если бит установлен
jnz    short loc_40104D
...
jmp    short loc_401063
loc_40104D:
...
loc_401063:
...
```

Хотя представленный фрагмент я и прокомментировал прямо в тексте, следует сделать еще ряд замечаний. Команда fcomp сравнивает два операнда, и результат сравнения отражается на трех флагах: C0, C2, C3, которые соответствуют битам 8, 9, 10 в слове состояния SW *(см. разд. 1.2.3)*. В табл. 3.3 представлены значения флагов для различных ситуаций сравнения.

Таблица 3.3. *Значения флагов сравнения*

Проверяемое условие	**Флаг C3**	**Флаг C2**	**Флаг C0**
ST(0)>src	0	0	0
ST(0)<src	0	0	1
ST(0)==src	1	0	0
Операнды несравнимы	1	1	1

Из таблицы следует, что если флаг C0==0, то это как раз соответствует условию >=. Отсюда JNZ — переход "если не 0" на фрагмент кода для печати сообщения, что переменная b — наибольшая. Некоторые компиляторы используют другой прием. Они копируют флаги состояния сопроцессора при помощи команды SAHF в регистр флагов. При этом флаг C0 копируется во флаг CF, флаг C2 в PF, флаг C3 в ZF. А далее можно сразу использовать команды условных переходов. В данном примере это будет команда JZ (вместо JNZ). Хорошо, а как быть, если проверяется строгое неравенство a>b. Согласно табл. 3.3, команда проверки будет TEST AX,41H.

Вложенные конструкции и логические связки

В реальном программировании внутри условных конструкций могут содержаться другие условные конструкции (листинг 3.52). Рассмотрим, как вложение отражается на исполняемом коде.

Листинг 3.52

```
//поиск максимального значения из трех чисел
#include <stdio.h>
void main()
{
  int a,b,c;
  scanf("%d",&a);
  scanf("%d",&b);
  scanf("%d",&c);
  if(a>b)
  {
    if(a>c)printf("%d\n",a);
    else printf("%d\n",c);
  }else
  {
    if(b>c)printf("%d\n",b);
    else printf("%d\n",c);
  }
}
```

Оказывается, компилятор строит структуру вложенных условных конструкций по той же схеме, что мы видели в листинге 3.49. Эта схема представлена в листинге 3.53[10].

[10] В круглых скобках я указал уровень вложенности.

Листинг 3.53

```
jl l1
// if (1)
// выполняется a>b
jl l4
// if (2)
// выполняется a>c
// выводим значение a
...
jmp l2
l4:
// else (2)
// не выполняется a>c, но выполняется a>b
// выводим значение c
...
l2:
// конец оператора if первого уровня
jmp l3
// начало оператора else первого уровня
l1:
// else (1)
// не выполняется a>b
jl l5
// if (2)
// выполняется b>c и не выполняется a>b
// выводим значение b
...
jmp l3
l5:
// else (2)
// не выполняется b>c и не выполняется a>b
// выводим значение c
...
l3:
//конец вложенной конструкции
```

Внимательно рассмотрим схему из листинга 3.53. Мы видим, что она четко проецируется на схему в исходной программе (листинг 3.52). Причем любая

полная условная конструкция легко преобразуется в неполную условную конструкцию просто отбрасыванием соответствующей команды безусловного перехода jmp.

В современных языках программирования вместо большого количества вложенных друг в друга условных конструкций используются логические связки: and и or, при помощи которых образуются составные условия. В реальном программировании такие составные условия могут оказаться очень сложными. В замечательной книге "Фундаментальные основы хакерства" Криса Касперски[11] для анализа исполняемого кода, получившегося из сложных условных конструкций, предлагается использовать диаграммную технику. Такой подход действительно может помочь восстановить исходную сложную условную конструкцию. На мой взгляд, однако, для того чтобы понять заложенные в исполняемом коде условия, не всегда требуется восстановить исходный код. Ведь, согласитесь же, что авторы программ используют логические связки не столько для лучшего понимания их программы, сколько для сокращения записи. А краткость записи скорее усложняет чтение, нежели улучшает понимание. Кроме этого, многие программисты наряду с использованием логических связок применяют и вложенные условные конструкции, что ставит под вопрос попытку восстановить истинную условную конструкцию.

Рассмотрим следующий пример (листинг 3.54).

Листинг 3.54

```
.text:00401000  _main    proc near     ; CODE XREF: start+16E?p
.text:00401000    var_C  = dword ptr -0Ch
.text:00401000    var_8  = dword ptr -8
.text:00401000    var_4  = dword ptr -4
.text:00401000      push    ebp
.text:00401001      mov     ebp, esp
.text:00401003      sub     esp, 0Ch
.text:00401006      lea     eax, [ebp+var_4]
.text:00401009      push    eax
.text:0040100A      push    offset unk_4080FC
.text:0040100F      call    _scanf
.text:00401014      add     esp, 8
.text:00401017      lea     ecx, [ebp+var_8]
.text:0040101A      push    ecx
.text:0040101B      push    offset unk_408100
```

[11] Касперски Крис. Фундаментальные основы хакерства. — М.: СОЛОН-Пресс, 2004.

```
.text:00401020      call    _scanf
.text:00401025      add     esp, 8
.text:00401028      lea     edx, [ebp+var_C]
.text:0040102B      push    edx
.text:0040102C      push    offset unk_408104
.text:00401031      call    _scanf
.text:00401036      add     esp, 8
.text:00401039      mov     eax, [ebp+var_4]
.text:0040103C      cmp     eax, [ebp+var_8]
.text:0040103F      jle     short loc_401047
.text:00401041      cmp     [ebp+var_8], 0
.text:00401045      jg      short loc_401055
.text:00401047  loc_401047:     ; CODE XREF: _main+3F?j
.text:00401047      mov     ecx, [ebp+var_4]
.text:0040104A      cmp     ecx, [ebp+var_C]
.text:0040104D      jz      short loc_401055
.text:0040104F      cmp     [ebp+var_C], 0
.text:00401053      jnz     short loc_401064
.text:00401055  loc_401055:     ; CODE XREF: _main+45?j
.text:00401055      push    offset aYes      ; "Yes!\n"
.text:0040105A      call    _printf
.text:0040105F      add     esp, 4
.text:00401062      jmp     short loc_401071
.text:00401064  loc_401064:     ; CODE XREF: _main+53?j
.text:00401064      push    offset aNo       ; "No!\n"
.text:00401069      call    _printf
.text:0040106E      add     esp, 4
.text:00401071  loc_401071:     ; CODE XREF: _main+62?j
.text:00401071      xor     eax, eax
.text:00401073      mov     esp, ebp
.text:00401075      pop     ebp
.text:00401076      retn
.text:00401076  _main    endp
```

Комментарий к листингу 3.54

Конечно, мы достаточно много уже видели подобных листингов и легко различим, что в нем используются три стековые переменные целого типа (зарезервировано 12 байтов и используются три однотипные переменные). Ввод значений с помощью библиотечной функции `scanf` для нас тоже не в

новинку. Для нас интересны условные переходы. Прежде всего, давайте посмотрим на код, как говорится, с высоты птичьего полета. И что же оттуда видно? Да прежде всего то, что все условные переходы сводятся к двум результатам: печать при помощи функции `printf` строки `"Yes"` (адрес `00401055`) или строки `"No"` (адрес `00401064`). А, следовательно, очень похоже на то, что мы имеем дело с полной условной конструкцией (`if...else`). Теперь просто соберем все условия, приводящие к первому и второму результату. Итак, на адрес `00401055` осуществляется переход, если `var_4>var_8` и `var_8>0`. Это условие явно претендует на условие типа `a>b&&b>0`, назовем его условием (1). Поскольку тот же результат получается и при выполнении других условий, но не выполнении условия (1), то можно предположить, что речь идет о связке "ИЛИ", впрочем, для понимания это учитывать совсем не обязательно. Итак, тот же результат получается, если выполняется условие `var_4=var_C`. Это будет условие (2). Наконец, опять тот же результат получается, если выполняется условие `var_C=0`. Это будет условие (3). Во всех остальных случаях выполняется фрагмент, начинающийся с адреса `00401064`. После наших рассуждений мы вполне можем записать условную конструкцию при помощи связок "И" и "ИЛИ". Однако понимания нам это совсем не прибавит, поскольку мы уже разобрались в логике функционирования данного фрагмента.

Итак, какой же вывод можно сделать из проведенных только что рассуждений? Вывод следующий: условия, связанные друг с другом при помощи логического "И", достаточно легко распознаются. Рассматривая их как единое условие, можно легко понять логику и других условий (связанных с первым при помощи логического "ИЛИ").

Условные конструкции без переходов

Надо сказать, что условные и безусловные переходы сбрасывают очередь команд, что в конечном итоге замедляет выполнение программы. Имейте это в виду, когда пишете программу на ассемблере. В тех случаях, когда можно обойтись без переходов, следует обходиться без них. Для этой цели можно использовать наборы команд `SETcc r/m` (условная установка первого бита) и `CMOVX` (условная пересылка данных). Эти команды вы можете найти в табл. 1.1. Оказывается, об этом "знают" и продвинутые компиляторы, к которым, в частности, относится компилятор Visual C++. К сожалению, все мои старания заставить его использовать какую-либо из команд условной пересылки не увенчалась успехом, тогда как командами условной установки битов, как оказалось, он пользуется довольно часто.

Идея использования команд условной установки первого бита байта до тривиальности проста. Пусть некоторый регистр, например, `EAX`, вначале содержит значение 0. Далее после команды сравнения `CMP` мы используем одну из команд условной установки бита, например, `SETLE` — установить, если

меньше. После этого выполним команду DEC EAX. Теперь в случае выполнения условия EAX содержит 0, а в противном случае — −1 (FFFFFFFFH). Далее используем тот факт, что если операнд равен нулю, то воздействие на него инструкции and с любым вторым операндом не изменит его содержимое. Рассмотрим, к примеру, следующий фрагмент:

```
.text:00401013      xor     eax, eax
.text:00401015      cmp     edx, 0Ah
.text:00401018      setle   al
.text:0040101B      dec     eax
.text:0040101C      and     eax, 0FFFFF200h
.text:00401021      add     eax, 1000h
```

Легко видеть, что в случае выполнения условия (EDX<=0) в регистре EAX после выполнения действия будет содержаться значение 1000H, а в случае невыполнения условия — значение 200H. На самом деле я привел исполняемый код, который эквивалентен оператору

```
if(a>10) b=0x200;
else b=0x1000;
```

и который создает компилятор Microsoft при условии, что мы установим опцию "создавать быстрый код".

Операторы выбора

Большое количество неполных условных операторов, расположенных друг за другом, принято заменять оператором выбора. В листинге 3.55 приведен типичный пример использования оператора выбора. Посмотрим, что компилятор Microsoft сделает с данным текстом (листинг 3.56).

Листинг 3.55

```
#include <stdio.h>
void main()
{
  char a;
  scanf("%c",&a);
  switch(a)
  {
    case 'A':
      printf("A\n");
      break;
```

```
    case 'B':
      printf("B\n");
      break;
    default:
      printf("?\n");
  }
}
```

Листинг 3.56

```
.text:00401000  _main    proc near  ; CODE XREF: start+16E?p
.text:00401000    var_8  = byte ptr -8
.text:00401000    var_1  = byte ptr -1
.text:00401000      push    ebp
.text:00401001      mov     ebp, esp
.text:00401003      sub     esp, 8
.text:00401006      lea     eax, [ebp-1]
.text:00401009      push    eax
.text:0040100A      push    offset unk_4080FC
.text:0040100F      call    _scanf
.text:00401014      add     esp, 8
.text:00401017      mov     cl, [ebp+var_1]
.text:0040101A      mov     [ebp+var_8], cl
.text:0040101D      cmp     [ebp+var_8], 41h
.text:00401021      jz      short loc_40102B
.text:00401023      cmp     [ebp+var_8], 42h
.text:00401027      jz      short loc_40103A
.text:00401029      jmp     short loc_401049
.text:0040102B  loc_40102B:      ; CODE XREF: _main+21?j
.text:0040102B      push    offset byte_408100
.text:00401030      call    _printf
.text:00401035      add     esp, 4
.text:00401038      jmp     short loc_401056
.text:0040103A  loc_40103A:      ; CODE XREF: _main+27?j
.text:0040103A      push    offset unk_408104
.text:0040103F      call    _printf
.text:00401044      add     esp, 4
.text:00401047      jmp     short loc_401056
.text:00401049  loc_401049:      ; CODE XREF: _main+29?j
```

```
.text:00401049      push    offset unk_408108
.text:0040104E      call    _printf
.text:00401053      add     esp, 4
.text:00401056  loc_401056:         ; CODE XREF: _main+38?j
.text:00401056          ; _main+47?j
.text:00401056      xor     eax, eax
.text:00401058      mov     esp, ebp
.text:0040105A      pop     ebp
.text:0040105B      retn
.text:0040105B  _main    endp
```

Комментарий к листингу 3.56

- Код интересен, прежде всего, некоторой избыточностью. В процедуре `main` определены две стековые переменные. Переменная `var_1`, судя по использованию ее с функцией `scanf`, предназначена для хранения реально определенной в тексте программы переменной `a` (листинг 3.55). Переменная `var_8` является вспомогательной, временной переменной. Она используется для сравнения в командах типа `cmp [ebp+var_8],42h`. Согласитесь, что вполне можно было бы обойтись и одной переменной `var_1`. Кстати, обратите внимание, что две однобайтовые переменные хранятся в соседних четырехбайтовых блоках. Это диктуется требованием выравнивания по границе 4 байта.
- Схема проверки условий довольно-таки тяжеловесна. Эта тяжеловесность связана с тем, что осуществляется проверка именно выполнения условия. Согласитесь, что более изящной была бы следующая схема:

```
   cmp  [ebp+var_8], 41h
   jnz  l1
   ...
   jmp  _break
l1:
   cmp  [ebp+var_8], 42h
   jnz  l2
   ...
   jmp  _break
l2:
   //default
   ...
_break:
```

Наконец, вы можете встретиться и с подходом, представленным в листинге 3.57. В частности, именно так обрабатывает оператор выбора компилятор Borland C++ 5.0, а также Delphi. Компилятор Microsoft C++ ведет себя подобным образом, если установить опцию "создавать быстрый код".

Листинг 3.57

```
   mov  dl, [ebp+var_1]
   sub  dl, 41h
   jz   l1
   dec  dl
   jz   l2
   jmp  l3
l1:
   ...
   jmp  l3
l2:
   ...
l3:
   ...
```

Из листинга 3.57 легко можно понять принцип подхода. Параметр оператора `switch` помещается в некоторую временную переменную, которая, в частности, может оказаться и регистром. Пусть значения, на равенство которым будет проверяться переменная, равны, соответственно, `a1`, `a2`, ..., `an` (в порядке возрастания). Проверка осуществляется следующим образом. Вначале от временной переменной отнимается `a1` и проверяется, не равно ли после этого значение переменной 0. Далее из переменной будут отниматься значения `a2-a1`, `a3-a2` и т. д. После каждого такого отнимания проверяется, не равна ли переменная нулю. Особенно эффективно данный подход работает, если все разности `a2-a1`, `a3-a2` и т. д. равны единице. В этом случае удобно использовать команду процессора `dec`.

3.2.3. Циклы

Циклический алгоритм в действительности является разновидностью ветвления, в котором, в зависимости от условия, осуществляется многократное выполнение некоторого фрагмента программы. Если говорить об ассемблере, то в цикле предполагается условный или безусловный переход назад, к инструкциям с меньшими адресами. Современные дизассемблеры и отладчики отслеживают такие переходы и отмечают их в дизассемблированном тексте.

Простые циклы

Рассмотрим возможные варианты построения циклов на языке ассемблера (листинг 3.58). Это позволит нам легко разобраться в том, как с циклами обращаются современные компиляторы.

Листинг 3.58

```
...
;довольно часто здесь стоит JMP L1
_beg:
;здесь, в частности, могут стоять команды изменения параметра цикла
...
CMP EAX,EBX ; или проверка какого-либо другого условия
JZ  _end    ; или любой другой условный переход за границы цикла
;здесь начинается тело
L1:
...         ;тело цикла — произвольное количество команд
JMP _beg
_end:
```

Комментарий к листингу 3.58

В листинге 3.58 можно видеть типичную структуру цикла, с которой вы встретитесь при изучении кода, созданного самыми различными компиляторами. Обратите внимание на возможное наличие перехода на начало тела цикла (`JMP L1`). Дело в том, что если такой переход имеется, то тело цикла выполнится, по крайней мере, один раз. Это соответствует идеологии цикла "до" (такой цикл называют еще *циклом с постусловием*). Если же переход в тело цикла отсутствует, то в общем случае тело может и не выполниться ни разу. Это соответствует идеологии цикла "пока" (такой цикл называют еще *циклом с предусловием*).

Теперь рассмотрим листинг 3.59.

Листинг 3.59

```
...
_beg:
;начало тела цикла
...
```

```
CMP EAX,EBX  ; или проверка какого-либо другого условия
JZ  _beg     ; или любой другой условный переход на начало цикла
...
```

Комментарий к листингу 3.59

В данном случае мы имеем типичный пример цикла "до". Однако было бы ошибочно считать, что тип цикла в языке высокого уровня будет автоматически соответствовать типу цикла в исполняемом коде. Мы уже неоднократно убеждались в том, что компиляторы не просто тупо переводят операторы языка высокого уровня в машинные инструкции, а творчески (иногда слово творчески следовало бы ставить в кавычки) переосмысливают их. Поэтому если вы в своей программе используете цикл "пока", но соотношение значений переменных заведомо таково, что цикл исполнится, по крайней мере, один раз, то компилятор запросто может использовать в машинном варианте цикл "до".

Что касается циклов `for`, которые имеются в основных алгоритмических языках, то это есть лишь разновидность цикла с предусловием. Просто в формировании условия участвует один или несколько параметров. Параметры же таких циклов при проходе тела цикла (или перед очередным проходом) получают некоторое неизменное приращение, знак которого может быть и положительным, и отрицательным. Наличие переменной, которая при каждом проходе получает некоторое постоянное положительное или отрицательное приращение, является важным признаком того, что мы имеем дело с циклом `for`.

После такого введения естественно перейти к конкретным примерам, что мы сейчас и сделаем.

Листинг 3.60 содержит простейший пример использования цикла `for`.

Листинг 3.60

```
#include <stdio.h>
void main()
{
  int b=10;
  for(int i=0; i<100; i++)
    printf("%d\n",b);
}
```

Посмотрим, что же будет в исполняемом коде, который создаст компилятор Visual C++, при условии отсутствия оптимизации (листинг 3.61).

Листинг 3.61

```
.text:00401000  _main    proc near    ; CODE XREF: start+16E?p
.text:00401000    var_8  = dword ptr -8
.text:00401000    var_4  = dword ptr -4
.text:00401000      push    ebp
.text:00401001      mov     ebp, esp
.text:00401003      sub     esp, 8
.text:00401006      mov     [ebp+var_4], 0Ah
.text:0040100D      mov     [ebp+var_8], 0
.text:00401014      jmp     short loc_40101F
.text:00401016  loc_401016:     ; CODE XREF: _main+36?j
.text:00401016      mov     eax, [ebp+var_8]
.text:00401019      add     eax, 1
.text:0040101C      mov     [ebp+var_8], eax
.text:0040101F  loc_40101F:       ; CODE XREF: _main+14?j
.text:0040101F      cmp     [ebp+var_8], 64h
.text:00401023      jge     short loc_401038
.text:00401025      mov     ecx, [ebp+var_4]
.text:00401028      push    ecx
.text:00401029      push    offset unk_4060FC
.text:0040102E      call    _printf
.text:00401033      add     esp, 8
.text:00401036      jmp     short loc_401016
.text:00401038  loc_401038:       ; CODE XREF: _main+23?j
.text:00401038      xor     eax, eax
.text:0040103A      mov     esp, ebp
.text:0040103C      pop     ebp
.text:0040103D      retn
.text:0040103D  _main    endp
```

Комментарий к листингу 3.61

- Прежде всего, обращаю ваше внимание на локальные переменные. `var_4` — это, очевидно, переменная `b`. А вот переменная `var_8` есть не что иное, как параметр цикла. Также обращаю ваше внимание на знаковые команды

  ```
  mov eax,[ebp+var_8]
  add eax,1
  mov [ebp+var_8],eax
  ```

  Они бросаются в глаза и являются очевидным признаком цикла `for`.

- Листинг 3.61 как раз соответствует схеме из листинга 3.58. Переход же `jmp short loc_40101F` отражает тот факт, что начальное значение параметра цикла должно быть равно 0, а условие выхода из цикла — равенство этого параметра 100. Разумеется, если бы мы писали подобный алгоритм на ассемблере, то могли бы просто обойтись без этого перехода, присвоив начальное значение параметру цикла −1.

Глядя на листинг 3.61, мы понимаем, что перед нами цикл "пока". Однако посмотрите, что выйдет, если сообщить компилятору, что мы желаем получить компактный исполняемый код (листинг 3.62).

Листинг 3.62

```
.text:00401000  _main     proc near       ; CODE XREF: start+16E?p
.text:00401000      push    esi
.text:00401001      push    64h
.text:00401003      pop     esi
.text:00401004  loc_401004:               ; CODE XREF: _main+13?j
.text:00401004      push    0Ah
.text:00401006      push    offset unk_4060FC
.text:0040100B      call    _printf
.text:00401010      dec     esi
.text:00401011      pop     ecx
.text:00401012      pop     ecx
.text:00401013      jnz     short loc_401004
.text:00401015      xor     eax, eax
.text:00401017      pop     esi
.text:00401018      retn
.text:00401018  _main     endp
```

Посмотрите на листинг 3.61 — перед нами типичный пример цикла "до". Обратите внимание, что роль параметра цикла играет регистр `ESI`. Причем вместо того чтобы увеличивать параметр, он уменьшает его так, чтобы использовать условие сравнения с 0. Это типичный прием, который применяет компилятор Visual C++: превратить цикл "пока" в цикл "до". При этом инкремент параметра цикла заменяется декрементом. Интересно, что если заменить цикл `for` на цикл `while` или цикл `do...while` и сохранить опцию оптимизации "создавать компактный код", то мы опять придем в точности к листингу 3.62.

Разберем теперь листинг 3.63.

Листинг 3.63

```
.text:00401108  _main    proc near  ; DATA XREF: .data:0040A0B8?o
.text:00401108    argc  = dword ptr  8
.text:00401108    argv  = dword ptr  0Ch
.text:00401108    envp  = dword ptr  10h
.text:00401108      push    ebp
.text:00401109      mov     ebp, esp
.text:0040110B      push    ebx
.text:0040110C      push    esi
.text:0040110D      mov     esi, 0Ah
.text:00401112      xor     ebx, ebx
.text:00401114  loc_401114:                     ; CODE XREF: _main+1E?j
.text:00401114      push    esi
.text:00401115      push    offset format   ; format
.text:0040111A      call    _printf
.text:0040111F      add     esp, 8
.text:00401122      inc     ebx
.text:00401123      cmp     ebx, 64h
.text:00401126      jl      short loc_401114
.text:00401128      pop     esi
.text:00401129      pop     ebx
.text:0040112A      pop     ebp
.text:0040112B      retn
.text:0040112B  _main    endp
```

Интересно, что ту же программу компилятор Borland C++ 5.0 трактует несколько иначе: цикл с предусловием преобразуется к циклу с постусловием, но инкремент параметра цикла остается. Кроме того, заметим, что компилятор Microsoft более аккуратно работает с регистрами — используется всего один регистр (`ESI`), требующий сохранения в стеке.

Об оптимизации циклов

Собственно, об одном из видов оптимизации мы уже говорили: преобразование цикла с предусловием в цикл с постусловием. Кроме этого, мы видели, что компилятор Microsoft может заменить инкремент параметра цикла его декрементом, так что условием выхода из цикла будет его равенство нулю. Но есть и более действенные способы оптимизации циклов, используемые продвинутыми компиляторами. Мы разберем два из них.

Вычисление на стадии компиляции

Довольно часто программист не замечает, что результат, который получается при выполнении циклического алгоритма, и так очевиден. Часто компиляторы, к каковым относится и компилятор Visual C++, распознают такие ситуации и заменяют циклические алгоритмы готовыми результатами. Рассмотрим следующий пример (листинг 3.64).

Листинг 3.64

```
#include <stdio.h>
void main()
{
  int i=0,s=0,k=5;
  for(i=0; i<k; i++)s=s+i;
  printf("%d\n",s);
}
```

В принципе, совсем не трудно сосчитать, каковой будет сумма `s` в результате вычисления. Она будет равна 10. Компилятор Visual C++ легко справляется с этой задачей. Посмотрите, что получается в результате компиляции с опцией "создавать быстрый код" (листинг 3.65).

Листинг 3.65

```
.text:00401000  _main    proc near    ; CODE XREF: start+16E?p
.text:00401000      push    0Ah
.text:00401002      push    offset unk_4060FC
.text:00401007      call    _printf
.text:0040100C      add     esp, 8
.text:0040100F      xor     eax, eax
.text:00401011      retn
.text:00401011  _main    endp
```

В листинге 3.65 никакого цикла нет и в помине. Лишнее подтверждение того, что процесс компиляции в общем случае не обратим, но к пониманию логики программы это не имеет отношения.

Разворачивание циклов

Разворачивание циклов — это прием, основанный на принципах оптимизации обращения к памяти.

В листинге 3.66 представлена программа, где используется массив a. Откомпилируем ее с помощью компилятора Visual C++, указав опцию "создавать быстрый код". Результат дизассемблирования исполняемого кода представлен в листинге 3.67.

Листинг 3.66

```
#include <stdio.h>
void main()
{
  int a[100];
  int i=0,s=0;
  for(i=0; i<100; i++)a[i]=i;
  for(i=0; i<100; i++)s+=a[i];
  printf("%d\n",s);
}
```

Листинг 3.67

```
.text:00401000  _main    proc near    ; CODE XREF: start+16E?p
.text:00401000    var_194    = dword ptr -194h
.text:00401000    var_190    = dword ptr -190h
.text:00401000    var_18C    = dword ptr -18Ch
.text:00401000    var_188    = dword ptr -188h
.text:00401000    var_184    = dword ptr -184h
.text:00401000    var_180    = dword ptr -180h
.text:00401000      sub     esp, 190h
.text:00401006      xor     ecx, ecx
.text:00401008      xor     eax, eax
.text:0040100A      lea     ebx, [ebx+0]
.text:00401010  loc_401010:         ; CODE XREF: _main+17?j
.text:00401010      mov     [esp+eax*4+190h+var_190], eax
.text:00401013      inc     eax
.text:00401014      cmp     eax, 64h
.text:00401017      jl      short loc_401010
.text:00401019      xor     eax, eax
.text:0040101B      push    esi
.text:0040101C      lea     esp, [esp+0]
.text:00401020  loc_401020:         ; CODE XREF: _main+3E?j
.text:00401020      mov     esi, [esp+eax*4+194h+var_184]
```

```
.text:00401024       mov     edx, [esp+eax*4+194h+var_180]
.text:00401028       add     edx, esi
.text:0040102A       add     edx, [esp+eax*4+194h+var_188]
.text:0040102E       add     edx, [esp+eax*4+194h+var_18C]
.text:00401032       add     edx, [esp+eax*4+194h+var_190]
.text:00401036       add     eax, 5
.text:00401039       add     ecx, edx
.text:0040103B       cmp     eax, 64h
.text:0040103E       jl      short loc_401020
.text:00401040       push    ecx
.text:00401041       push    offset unk_4060FC
.text:00401046       call    _printf
.text:0040104B       add     esp, 8
.text:0040104E       xor     eax, eax
.text:00401050       pop     esi
.text:00401051       add     esp, 190h
.text:00401057       retn
.text:00401057  _main     endp
```

Комментарий к листингу 3.67

- Для локального массива отводится 190h байтов, что как раз равно 400, т. е. по 4 байта на каждый элемент.
- Сразу обращу ваше внимание на две команды: lea ebx,[ebx] и lea esp,[esp]. Разумеется, обе команды не меняют содержимое регистров и помещены компилятором для оптимизации скорости выполнения. Ранее мы уже говорили о таком приеме оптимизации, как спаривание команд (см. комментарий к листингу 3.2). В данном случае компилятор добавляет ничего не значащие команды для правильного разбиения команд по парам.
- Цикл, в котором задаются значения элементов массива, достаточно прост. Он располагается в листинге по адресам 00401010—00401017. Обратим внимание, что адресация локальных переменных в стеке осуществляется не посредством регистра EBP, который здесь из целей оптимизации вообще не используется, а посредством регистра ESP. Регистр EAX играет роль переменной i (пример использования регистровой переменной). Замечу, что в данном цикле компилятор не использует прием замены инкремента параметра цикла на декремент, о котором мы говорили ранее. Это связано просто с тем, что параметр цикла играет также и роль индекса в массиве.

- Обратимся теперь к циклу, располагающемуся по адресам 00401020—0040103E, назначение которого — просуммировать все элементы массива. Сумма должна накапливаться в переменной s (см. листинг 3.66). Легко видеть, что роль переменной s играет регистр ECX (это видно хотя бы по вызову к функции printf). Очень важная деталь: перед циклом стоит команда PUSH ESI. Причина понятна — ведь ESI используется далее как временная переменная, а этот регистр не должен измениться после выполнения функции main. Но не забывайте, что адресация идет относительно регистра ESP, и далее компилятор корректирует адресацию элементов массива. Таким образом, при вычислении индекса мы должны вычитать единицу (4 при вычислении адреса элемента). Итак, имеем:
  - [esp+eax*4+194h+var_184] = [esp+eax*4+4*4], что соответствует a[i+3];
  - [esp+eax*4+194h+var_180] = [esp+eax*4+5*4], что соответствует a[i+4];
  - [esp+eax*4+194h+var_188] = [esp+eax*4+3*4], что соответствует a[i+2];
  - [esp+eax*4+194h+var_18C] = [esp+eax*4+2*4], что соответствует a[i+1];
  - [esp+eax*4+194h+var_190] = [esp+eax*4+1*4], что соответствует a[i].

  Что же у нас получается? Вначале складываются элементы a[i+3] и a[i+4] и результат помещается в регистр EDX. Далее к сумме добавляются элементы a[i+2], a[i+1] и a[i]. Окончательная сумма добавляется к регистру ECX, который, как мы знаем, и является переменной s. Наконец индекс, т. е. параметр цикла, разумеется, увеличивается не на 1, а на 5. Таким образом, можно сказать, что компилятор, в сущности, реализовал следующий цикл:

```
for(i=0; i<100; i+=5)
{
  s=s+a[i];
  s=s+a[i+1];
  s=s+a[i+2];
  s=s+a[i+3];
  s=s+a[i+4];
}
```

  Безусловно, данный цикл с алгоритмической точки зрения полностью эквивалентен циклу из листинга 3.66. С точки же зрения оптимизации он реализует так называемое ***разворачивание циклов***, позволяющее значительно увеличить скорость выполнения программы.

Вложенные циклы и циклы со сложными условиями выхода

Выше мы разбирали структуры исполняемого кода, соответствующие простейшим циклам. Однако циклические алгоритмы могут быть усложнены следующими факторами:

- вложенностью циклов;
- сложным условием выхода из цикла;
- наличием дополнительных операторов управления циклами, такими как `break` и `continue`.

Приведу только два примера. Первый пример содержит в себе одновременно и сложное условие выхода из цикла, и операторы управления циклами (листинг 3.68). Я намеренно не привожу исходный текст программы на C++, поскольку нашей задачей является не посмотреть, как компилятор преобразует текст программы в исполняемый код, а понять логику выполнения этого кода.

Листинг 3.68

```
.text:00401000  _main    proc near     ; CODE XREF: start+16E?p
.text:00401000    var_C  = dword ptr -0Ch
.text:00401000    var_8  = dword ptr -8
.text:00401000    var_4  = dword ptr -4
.text:00401000      push    ebp
.text:00401001      mov     ebp, esp
.text:00401003      sub     esp, 0Ch
.text:00401006      mov     [ebp+var_4], 0
.text:0040100D      mov     [ebp+var_C], 0
.text:00401014      mov     [ebp+var_8], 0
.text:0040101B      jmp     short loc_401026
.text:0040101D  loc_40101D:            ; CODE XREF: _main+51?j
.text:0040101D                 ; _main+74?j
.text:0040101D      mov     eax, [ebp+var_8]
.text:00401020      add     eax, 1
.text:00401023      mov     [ebp+var_8], eax
.text:00401026  loc_401026:            ; CODE XREF: _main+1B?j
.text:00401026      cmp     [ebp+var_8], 2710h
.text:0040102D      jge     short loc_401076
.text:0040102F      cmp     [ebp+var_4], 0C350h
.text:00401036      jge     short loc_401076
```

```
.text:00401038      mov     ecx, [ebp+var_4]
.text:0040103B      add     ecx, [ebp+var_C]
.text:0040103E      add     ecx, [ebp+var_8]
.text:00401041      mov     [ebp+var_4], ecx
.text:00401044      mov     [ebp+var_4], 1Eh
.text:0040104B      cmp     [ebp+var_4], 0
.text:0040104F      jz      short loc_401053
.text:00401051      jmp     short loc_40101D
.text:00401053  loc_401053:             ; CODE XREF: _main+4F?j
.text:00401053      mov     eax, [ebp+var_4]
.text:00401056      cdq
.text:00401057      idiv    [ebp+var_8]
.text:0040105A      mov     [ebp+var_C], eax
.text:0040105D      cmp     [ebp+var_C], 64h
.text:00401061      jnz     short loc_401065
.text:00401063      jmp     short loc_401076
.text:00401065  loc_401065:             ; CODE XREF: _main+61?j
.text:00401065      mov     edx, [ebp+var_C]
.text:00401068      mov     [ebp+var_C], edx
.text:0040106B      mov     eax, [ebp+var_C]
.text:0040106E      add     eax, 1
.text:00401071      mov     [ebp+var_C], eax
.text:00401074      jmp     short loc_40101D
.text:00401076  loc_401076:             ; CODE XREF: _main+2D?j
.text:00401076          ; _main+36?j ...
.text:00401076      xor     eax, eax
.text:00401078      mov     esp, ebp
.text:0040107A      pop     ebp
.text:0040107B      retn
.text:0040107B  _main    endp
```

Комментарий к листингу 3.68

Начнем с фрагмента, расположенного по адресам 0040101D—00401023. Очевидно перед нами типичный заголовок цикла, точнее, та часть цикла, которая увеличивает значение параметра цикла. Соответственно вход в цикл осуществляется ниже данного фрагмента по команде jmp short loc_401026. Подобные фрагменты мы изучали уже неоднократно. Итак, начало цикла очерчивается достаточно четко: это метка loc_40101D. Просматривая листинг далее, обратим внимание на команду jmp short loc_40101D.

Поскольку далее нет команд, осуществляющих переход на метки между адресами 0040101D и 00401074, то можно предположить, что это последняя команда цикла. Замечу, что, в принципе, рассматриваемый цикл мог быть вложенным и в другой цикл, но наш анализ не касается этого вопроса. Итак, контуры цикла очерчены. Следующий вопрос: по каким условиям происходит выход из цикла? Вообще, нам совершенно не важно, записано это условие в заголовке или выход осуществляется по оператору break. Важно, что выход из цикла происходит на адрес 00401076. Итак, мы видим следующие команды:

```
.text:0040102D      jge short loc_401076
...
.text:00401036      jge short loc_401076
...
text:00401063       jmp short loc_401076
```

Последний переход стоит несколько ниже двух первых и происходит при условии, что переменная var_C равна 64h, т. е. 100. По-видимому, это и есть выход по break. Первые же два перехода, очевидно, соответствуют заголовку цикла и условию логического "И". Без труда можно определить, что условие выполнения цикла может быть записано как var_8 <2710h && var_4<c350h. Замечу, что, очевидно, var_8 является параметром цикла (листинг 3.68, адреса 0040101D—00401023). По сути, неразъясненными в структуре цикла остались команды:

```
.text:0040104B      cmp [ebp+var_4], 0
.text:0040104F      jz  short loc_401053
.text:00401051      jmp short loc_40101D
```

Другими словами, если переменная var_4 окажется не равной нулю, то произойдет переход на начало цикла. Ну, а это, очевидно, есть не что иное, как оператор continue.

Рассмотрим теперь пример *вложенных циклов*. Типичным примером вложенных циклов является манипуляция с многомерными массивами. В листинге 3.69 представлен дизассемблированный код, который осуществляет заполнение двумерного массива. Замечу, что при компиляции программы при помощи компилятора Visual C++ я не задавал никаких опций оптимизации.

Листинг 3.69

```
.text:00401000  _main    proc near     ; CODE XREF: start+16E?p
.text:00401000    var_198    = dword ptr -198h
.text:00401000    var_194    = dword ptr -194h
```

```
.text:00401000   var_190    = dword ptr -190h
.text:00401000     push     ebp
.text:00401001     mov      ebp, esp
.text:00401003     sub      esp, 198h
.text:00401009     mov      [ebp+var_194], 0
.text:00401013     jmp      short loc_401024
.text:00401015  loc_401015:      ; CODE XREF: _main:loc_401078?j
.text:00401015     mov      eax, [ebp+var_194]
.text:0040101B     add      eax, 1
.text:0040101E     mov      [ebp+var_194], eax
.text:00401024  loc_401024:      ; CODE XREF: _main+13?j
.text:00401024     cmp      [ebp+var_194], 0Ah
.text:0040102B     jge      short loc_40107A
.text:0040102D     mov      [ebp+var_198], 0
.text:00401037     jmp      short loc_401048
.text:00401039  loc_401039:      ; CODE XREF: _main+76?j
.text:00401039     mov      ecx, [ebp+var_198]
.text:0040103F     add      ecx, 1
.text:00401042     mov      [ebp+var_198], ecx
.text:00401048  loc_401048:      ; CODE XREF: _main+37?j
.text:00401048     cmp      [ebp+var_198], 0Ah
.text:0040104F     jge      short loc_401078
.text:00401051     mov      edx, [ebp+var_194]
.text:00401057     add      edx, [ebp+var_198]
.text:0040105D     mov      eax, [ebp+var_194]
.text:00401063     imul     eax, 28h
.text:00401066     lea      ecx, [ebp+eax+var_190]
.text:0040106D     mov      eax, [ebp+var_198]
.text:00401073     mov      [ecx+eax*4], edx
.text:00401076     jmp      short loc_401039
.text:00401078  loc_401078:      ; CODE XREF: _main+4F?j
.text:00401078     jmp      short loc_401015
.text:0040107A  loc_40107A:      ; CODE XREF: _main+2B?j
.text:0040107A     xor      eax, eax
.text:0040107C     mov      esp, ebp
.text:0040107E     pop      ebp
.text:0040107F     retn
.text:0040107F  _main    endp
```

Комментарий к листингу 3.69

- Увидеть вложенные циклы в листинге достаточно просто. Легко различаются начало внешнего цикла по адресу `00401015` и начало внутреннего цикла по адресу `00401039`. Обращаю ваше внимание на инструкции `jmp short loc_401024` и `jmp short loc_401048`, которые осуществляют первичный вход, соответственно, во внешний и внутренний циклы. Таким образом, структура вложенных циклов в данном случае весьма проста и не требует дополнительных пояснений.
- Интересно рассмотреть то, как в исполняемом коде реализован алгоритм присвоения значения двумерному массиву. Итак, параметр внешнего цикла хранится в переменной `var_194`, а параметр внутреннего цикла — в переменной `var_198`. Оставшаяся переменная `var_190`, очевидно, указывает на начало нашего двумерного массива. Итак, после команд `mov edx,[ebp+var_194]` и `add edx,[ebp+var_198]` мы имеем в регистре `EDX` сумму значений двух индексов. Если забежать несколько вперед к команде `mov [ecx+eax*4],edx`, которая явно напоминает оператор присвоения значения элементу массива, то очевидно, что элементам массива присваиваются значения суммы индексов (значений параметров внешнего и внутреннего цикла). Но вернемся несколько назад. Что значат команды

```
mov  eax, [ebp+var_194]
imul eax, 28h
```

  Очевидно, что значение индекса (будем считать его первым) умножается на количество байтов в строке двумерного массива ($4 \times 10 = 40 = 28h$). А далее — `lea ecx,[ebp+eax+var_190]` — мы получаем адрес начала текущей строки в регистр `ECX`. Наконец, `ecx+eax*4` — это уже адрес текущего элемента двумерного массива. Таким образом, мы достаточно легко раскусили представленный в листинге 3.69 алгоритм.

Завершая разговор о циклах, замечу, что если при компиляции той же программы (с двумерным массивом) при помощи Visual C++ установить опцию "создавать быстрый код", то результат будет весьма интересен: исчезнет внутренний цикл, компилятор "развернет" его (см. листинг 3.67 и комментарий к нему). Внешний же цикл, как и следовало, из цикла с предусловием превратится в цикл с постусловием.

3.2.4. Объекты

Идентификация объектов и всего, что с ними связано, — задача более сложная, чем те, которые мы с вами до сих пор решали. Однако с большей частью материала, который будет изложен далее, мы уже встречались. Ведь методы, в конце концов, — это функции, а свойства объектов — это переменные[12]. Ну, а о деталях по порядку.

---

[12] Те и другие называют *членами класса*.

Идентификация объекта

Статические объекты

Начнем рассмотрение объектов с простого примера, который призван продемонстрировать нам типичные программные структуры обслуживания объектов (листинг 3.70). В программе имеется всего один класс, от которого создается единственный объект. Замечу, что создается *глобальный статический объект*, т. е. компилятор должен позаботиться о выделении памяти для него. Вообще центральным моментом в объектном программировании является вопрос о том, относятся ли созданные свойства и методы к одному объекту или нескольким объектам сразу. И если, например, метод (т. е. в сущности, некая функция) относится к нескольким объектам сразу, то как он (метод) "знает", относительно какого объекта он вызван в том или ином месте программы?

Листинг 3.70

```
#include <stdio.h>
class A {
public:
  int b;
  int a;
  int geta(){b=0; return a;};
  void seta(int);
};
void A::seta(int a1)
{
  a=a1;
  b=1;
};
A A1;
void main()
{
  A1.seta(10);
  int c=A1.geta();
  printf("%d\n",c);
}
```

В листинге 3.71 мы имеем дизассемблированный IDA Pro исполняемый код функции `main` для программы из листинга 3.70. Компилирование произведено в Visual C++ при отсутствии опций оптимизации.

Листинг 3.71

```
.text:00401020  _main    proc near    ; CODE XREF: start+16E?p
.text:00401020    var_4  = dword ptr -4
.text:00401020      push    ebp
.text:00401021      mov     ebp, esp
.text:00401023      push    ecx
.text:00401024      push    0Ah
.text:00401026      mov     ecx, offset unk_4086C0
.text:0040102B      call    sub_401000
.text:00401030      mov     ecx, offset unk_4086C0
.text:00401035      call    sub_401060
.text:0040103A      mov     [ebp+var_4], eax
.text:0040103D      mov     eax, [ebp+var_4]
.text:00401040      push    eax
.text:00401041      push    offset unk_4060FC
.text:00401046      call    _printf
.text:0040104B      add     esp, 8
.text:0040104E      xor     eax, eax
.text:00401050      mov     esp, ebp
.text:00401052      pop     ebp
.text:00401053      retn
.text:00401053  _main    endp
```

Комментарий к листингу 3.71

В тексте, представленном в листинге, имеются вызовы трех функций. Один вызов — это вызов библиотечной функции `printf`, и на нем нет смысла останавливаться. А вот функции `sub_401000` и `sub_401060` стоит рассмотреть подробнее. Ниже в листинге 3.72 мы разберем код этих функций. Поскольку у нас перед глазами текст программы (см. листинг 3.70), то, не мудрствуя лукаво, можно с определенностью сказать, что перед нами вызовы методов `seta` и `geta` соответственно. Обратим внимание, что перед вызовом метода в обоих случаях в регистр `ECX` отправляется адрес некой области памяти: `offset unk_4086C0`. Щелкнув по ссылке, мы узнаем, что область эта состоит из восьми байтов, во всяком случае, так сообщает нам IDA Pro. Это должно насторожить нас, ведь наш объект содержит два свойства, имеющих тип `int`, это как раз и составляет 8 байтов. Таким образом, уже из предварительных рассуждений можно сделать вывод, что при вызове метода одним из параметров является (для компиля-

тора Visual C++ параметр передается через регистр ECX) адрес объекта, относительно которого вызывается данный метод.

Замечание

Указатель на объект, используемый в методе, в языке C++ имеет свое название — `this`. С помощью данного указателя можно получить доступ к членам класса, в том числе и закрытым.

Листинг 3.72

```
.text:00401000  sub_401000  proc near    ; CODE XREF: _main+B?p
.text:00401000    var_4  = dword ptr -4
.text:00401000    arg_0  = dword ptr  8
.text:00401000      push     ebp
.text:00401001      mov      ebp, esp
.text:00401003      push     ecx
.text:00401004      mov      [ebp+var_4], ecx
.text:00401007      mov      eax, [ebp+var_4]
.text:0040100A      mov      ecx, [ebp+arg_0]
.text:0040100D      mov      [eax+4], ecx
.text:00401010      mov      edx, [ebp+var_4]
.text:00401013      mov      dword ptr [edx], 1
.text:00401019      mov      esp, ebp
.text:0040101B      pop      ebp
.text:0040101C      retn     4
.text:0040101C  sub_401000  endp

.text:00401060  sub_401060    proc near  ; CODE XREF: _main+15?p
.text:00401060    var_4  = dword ptr -4
.text:00401060      push     ebp
.text:00401061      mov      ebp, esp
.text:00401063      push     ecx
.text:00401064      mov      [ebp+var_4], ecx
.text:00401067      mov      eax, [ebp+var_4]
.text:0040106A      mov      dword ptr [eax], 0
.text:00401070      mov      ecx, [ebp+var_4]
.text:00401073      mov      eax, [ecx+4]
.text:00401076      mov      esp, ebp
```

```
.text:00401078          pop     ebp
.text:00401079          retn
.text:00401079 sub_401060 endp
```

Комментарий к листингу 3.72

- Рассмотрим текст функции sub_401000. Как мы уже заметили, очевидно, что это функция seta. Отмечу сразу, что функция имеет одну стековую переменную и один параметр. При этом освобождается стек посредством команды RETN 4. Привлекает внимание последовательность команд

  ```
  push ecx
  mov  [ebp+var_4],ecx
  ```

  Конечно, это недоразумение. Команда push здесь используется для резервирования памяти под локальную переменную. Одновременно переменной присваивается значение, содержащееся в регистре ECX. Следующая за push команда опять присваивает переменной то же значение. Извиним компилятор за такой казус — в конце концов, мы указали ему не проводить оптимизацию. Далее дело техники: значение, которое мы считаем адресом объекта, оказывается в регистре EAX и далее mov [eax+4],ecx (в ECX теперь значение параметра). Другими словами, свойство a объекта находится со сдвигом в 4 байта относительно адреса начала объекта. Далее:

  ```
  mov edx,[ebp+var_4]
  mov dword ptr [edx],1
  ```

  А это уже присвоение свойству b значения 1.

- После наших исследований в процедуре sub_401060 уже нет ничего нового. И mov eax,[ecx+4] — это просто возвращение значения свойства a функций geta.

Итак, рассмотрев пример компиляции при помощи Visual C++, можно отметить, что при вызове метода ему неявно передается указатель на объект, в контексте которого он вызывается. В нашем примере создавался глобальный объект. Однако нет никакой разницы, если объект будет создан локально, в стеке. Попробуйте сами провести это исследование.

Нет существенной разницы в том, как компилирует текст программы из листинга 3.70 компилятор Borland 5.0. Методу также сообщается адрес объекта посредством дополнительного параметра, который передается через стек последним по отношению к другим параметрам. Опять же, мы не будем обременять вас, дорогие читатели, дополнительными листингами, которые уже не привнесут в ваше понимание ничего нового.

Динамические объекты

При практическом программировании чаще используются объекты, создаваемые "на лету". Для этого применяется оператор `new`. Перепишем нашу программу из листинга 3.70 в терминах динамически создаваемых объектов (листинг 3.73).

Листинг 3.73

```
#include <stdio.h>
class A {
public:
  int b;
  int a;
  int geta(){b=0; return a;};
  void seta(int);
};
void A::seta(int a1)
{
  a=a1;
  b=1;
};
void main()
{
  A * A1=new(A);
  A1->seta(10);
  int c=A1->geta();
  printf("%d\n",c);
  delete A1;
}
```

В листинге 3.74 представлен исполняемый код, созданный компилятором Visual C++ в отсутствие оптимизации.

Листинг 3.74

```
.text:00401020  _main    proc near    ; CODE XREF: start+16E?p
.text:00401020    var_10  = dword ptr -10h
.text:00401020    var_C  = dword ptr -0Ch
.text:00401020    var_8  = dword ptr -8
```

```
.text:00401020    var_4  = dword ptr -4
.text:00401020      push    ebp
.text:00401021      mov     ebp, esp
.text:00401023      sub     esp, 10h
.text:00401026      push    8
.text:00401028      call    ??2@YAPAXI@Z      ; operator new(uint)
.text:0040102D      add     esp, 4
.text:00401030      mov     [ebp+var_C], eax
.text:00401033      mov     eax, [ebp+var_C]
.text:00401036      mov     [ebp+var_8], eax
.text:00401039      push    0Ah
.text:0040103B      mov     ecx, [ebp+var_8]
.text:0040103E      call    sub_401000
.text:00401043      mov     ecx, [ebp+var_8]
.text:00401046      call    sub_401080
.text:0040104B      mov     [ebp+var_4], eax
.text:0040104E      mov     ecx, [ebp+var_4]
.text:00401051      push    ecx
.text:00401052      push    offset unk_4060FC
.text:00401057      call    _printf
.text:0040105C      add     esp, 8
.text:0040105F      mov     edx, [ebp+var_8]
.text:00401062      mov     [ebp+var_10], edx
.text:00401065      mov     eax, [ebp+var_10]
.text:00401068      push    eax
.text:00401069      call    j__free
.text:0040106E      add     esp, 4
.text:00401071      xor     eax, eax
.text:00401073      mov     esp, ebp
.text:00401075      pop     ebp
.text:00401076      retn
.text:00401076    _main    endp
```

Комментарий к листингу 3.74

Конечно, не может не удивлять обилие во фрагменте и лишнего кода, и лишних стековых переменных. В данном случае это не имеет существенного значения. Важно, что после создания объекта дальнейшие действия, в сущности, ничем не отличаются от тех, что мы видели в листинге 3.71.

Замечу, что операторы new и delete (вызов процедуры j__free) распознаются дизассемблером IDA Pro, что делает дальнейший анализ весьма простым делом.

Виртуальные функции

Виртуальные функции — это своего рода плата за красоту и стройность теории объектного программирования. "Платит по счетам", разумеется исполняемый код и те, кто его исследуют, т. е. мы с вами.

Рассмотрим листинг 3.75.

Листинг 3.75

```
#include <stdio.h>
class A {
public:
  int a;
  int seta(int a1){a=a1; return a;};
  void pa(){printf("%d\n",a);}
};
class B:public A {
public:
  int seta(int a1){a=a1+1; return a;};
};
void main()
{
  A* A1;
  A1=new(B);
  A1->seta(10);
  A1->pa();
  delete A1;
};
```

В листинге 3.75 мы имеем типичный пример *наследования*. Класс B наследует свойства и методы класса A. При этом класс B имеет метод seta, совпадающий по имени и параметрам с подобным методом в классе A. Если, например, создать объект на основе класса B, то при вызове метода seta будет вызываться именно метод из класса B. Это известное свойство наследования. Несколько иная ситуация возникает, если создать вначале указатель на объект базового класса A, а затем создать объект при помощи оператора new

на основе шаблона класса в (см. листинг 3.75). Здесь компилятор по вполне очевидным причинам будет ориентироваться на тип указателя. Тогда `A1->seta(10)` будет означать вызов метода базового класса.

В листинге 3.76 представлен дизассемблированный исполняемый код, созданный Visual C++ при отсутствии оптимизации.

Листинг 3.76

```
.text:00401000  _main    proc near    ; CODE XREF: start+16E?p
.text:00401000    var_C  = dword ptr -0Ch
.text:00401000    var_8  = dword ptr -8
.text:00401000    var_4  = dword ptr -4
.text:00401000      push    ebp
.text:00401001      mov     ebp, esp
.text:00401003      sub     esp, 0Ch
.text:00401006      push    4
.text:00401008      call    ??2@YAPAXI@Z    ; operator new(uint)
.text:0040100D      add     esp, 4
.text:00401010      mov     [ebp+var_8], eax
.text:00401013      mov     eax, [ebp+var_8]
.text:00401016      mov     [ebp+var_4], eax
.text:00401019      push    0Ah
.text:0040101B      mov     ecx, [ebp+var_4]
.text:0040101E      call    sub_401050
.text:00401023      mov     ecx, [ebp+var_4]
.text:00401026      call    sub_401070
.text:0040102B      mov     ecx, [ebp+var_4]
.text:0040102E      mov     [ebp+var_C], ecx
.text:00401031      mov     edx, [ebp+var_C]
.text:00401034      push    edx
.text:00401035      call    j__free
.text:0040103A      add     esp, 4
.text:0040103D      xor     eax, eax
.text:0040103F      mov     esp, ebp
.text:00401041      pop     ebp
.text:00401042      retn
.text:00401042  _main    endp
.text:00401042
.text:00401050
```

```
.text:00401050  sub_401050   proc near  ; CODE XREF: _main+1E?p
.text:00401050    var_4  = dword ptr -4
.text:00401050    arg_0  = dword ptr  8
.text:00401050      push    ebp
.text:00401051      mov     ebp, esp
.text:00401053      push    ecx
.text:00401054      mov     [ebp+var_4], ecx
.text:00401057      mov     eax, [ebp+var_4]
.text:0040105A      mov     ecx, [ebp+arg_0]
.text:0040105D      mov     [eax], ecx
.text:0040105F      mov     edx, [ebp+var_4]
.text:00401062      mov     eax, [edx]
.text:00401064      mov     esp, ebp
.text:00401066      pop     ebp
.text:00401067      retn    4
.text:00401067  sub_401050  endp
.text:00401067
.text:00401070
.text:00401070  sub_401070  proc near  ; CODE XREF: _main+26?p
.text:00401070    var_4  = dword ptr -4
.text:00401070      push    ebp
.text:00401071      mov     ebp, esp
.text:00401073      push    ecx
.text:00401074      mov     [ebp+var_4], ecx
.text:00401077      mov     eax, [ebp+var_4]
.text:0040107A      mov     ecx, [eax]
.text:0040107C      push    ecx
.text:0040107D      push    offset unk_4060FC
.text:00401082      call    _printf
.text:00401087      add     esp, 8
.text:0040108A      mov     esp, ebp
.text:0040108C      pop     ebp
.text:0040108D      retn
.text:0040108D  sub_401070  endp
```

Комментарий к листингу 3.76

Комментарий будет коротким, т. к. в листинге мы не увидим принципиально ничего нового, отличного от того, что уже видели и исследовали в предыдущем разделе (см. листинг 3.74). Для создания нового объекта отводится

4 байта, а указатель на созданный объект передается через регистр ECX (см. вызов функций sub_401050 и sub_401070). В самих функциях указатель на объект используется для доступа к свойству объекта. В процедуре sub_401050 мы видим

```
mov  [ebp+var_4],ecx   ;адрес объекта в стековую переменную
mov  eax,[ebp+var_4]   ;адрес в регистр EAX
mov  ecx,[ebp+arg_0]   ;значение параметра в регистр ECX
mov  [eax],ecx         ;значение параметра присваивается свойству объекта
```

А теперь, собственно, перейдем к понятию виртуальных функций и полиморфизму. Для этого рассмотрим следующую программу из листинга 3.77. По сравнению с листингом 3.75 я добавил в базовый класс еще один метод seta1, а также сделал методы seta и seta1 виртуальными. Знакомые с объектным программированием сразу сообразят, что при вызовах A1->seta(10) и A1->seta1(10) будут вызываться методы производного класса, т. е. класса B.

Листинг 3.77

```
#include <stdio.h>
class A {
public:
  int a;
  virtual int seta(int a1){a=a1; return a;};
  virtual int seta1(int a1){a=2*a1; return a;};
  void pa(){printf("%d\n",a);}
};
class B:public A {
public:
  int seta(int a1){a=a1+1; return a;};
  int seta1(int a1){a=2*a1+1; return a;};
};
void main()
{
  A* A1;
  A1=new(B);
  A1->seta(10);
  A1->pa();
  A1->seta1(10);
```

```
  A1->pa();
  delete A1;
};
```

В листинге 3.78 представлен дизассемблированный исполняемый код функции `main` из листинга 3.77. Оптимизация отсутствует.

Листинг 3.78

```
.text:00401000  _main    proc near    ; CODE XREF: start+16E?p
.text:00401000    var_10  = dword ptr -10h
.text:00401000    var_C  = dword ptr -0Ch
.text:00401000    var_8  = dword ptr -8
.text:00401000    var_4  = dword ptr -4
.text:00401000      push    ebp
.text:00401001      mov     ebp, esp
.text:00401003      sub     esp, 10h
.text:00401006      push    8
.text:00401008      call    ??2@YAPAXI@Z    ; operator new(uint)
.text:0040100D      add     esp, 4
.text:00401010      mov     [ebp+var_8], eax
.text:00401013      cmp     [ebp+var_8], 0
.text:00401017      jz      short loc_401026
.text:00401019      mov     ecx, [ebp+var_8]
.text:0040101C      call    sub_4010A0
.text:00401021      mov     [ebp+var_10], eax
.text:00401024      jmp     short loc_40102D
.text:00401026  loc_401026:       ; CODE XREF: _main+17?j
.text:00401026      mov     [ebp+var_10], 0
.text:0040102D  loc_40102D:       ; CODE XREF: _main+24?j
.text:0040102D      mov     eax, [ebp+var_10]
.text:00401030      mov     [ebp+var_4], eax
.text:00401033      push    0Ah
.text:00401035      mov     ecx, [ebp+var_4]
.text:00401038      mov     edx, [ecx]
.text:0040103A      mov     ecx, [ebp+var_4]
.text:0040103D      call    dword ptr [edx]
.text:0040103F      mov     ecx, [ebp+var_4]
.text:00401042      call    sub_401080
```

```
.text:00401047      push    0Ah
.text:00401049      mov     eax, [ebp+var_4]
.text:0040104C      mov     edx, [eax]
.text:0040104E      mov     ecx, [ebp+var_4]
.text:00401051      call    dword ptr [edx+4]
.text:00401054      mov     ecx, [ebp+var_4]
.text:00401057      call    sub_401080
.text:0040105C      mov     eax, [ebp+var_4]
.text:0040105F      mov     [ebp+var_C], eax
.text:00401062      mov     ecx, [ebp+var_C]
.text:00401065      push    ecx
.text:00401066      call    j__free
.text:0040106B      add     esp, 4
.text:0040106E      xor     eax, eax
.text:00401070      mov     esp, ebp
.text:00401072      pop     ebp
.text:00401073      retn
.text:00401073  _main    endp
```

Комментарий к листингу 3.78

- В первую очередь обратим внимание на то, что при создании объекта запрашивается память не 4 байта как раньше, а 8 байтов (`push 8`). Забегая вперед, скажу, что дополнительные 4 байта, расположенные в начале объекта, отводятся для адреса таблицы виртуальных функций.
- На этот раз в тексте мы видим контроль того, что оператором `new` действительно была выделена память для создаваемого объекта. В случае если функция выделения памяти возвратит 0, т. е. произошла ошибка, в переменную `var_10` заносится 0, что в конечном итоге должно вызвать исключение (команда `mov edx, [ecx]` при содержимом регистра `ECX`, равном 0).
- Отдельно обрщаю ваше внимание на функцию `sub_4010A0` (листинг 3.79). Это особая функция, которой нет в тексте программы. Ее назначение — занести в начало области, отведенной для создаваемого объекта, адрес *таблицы виртуальных функций*. Функция возвращает, опять же, адрес объекта, но уже с адресом (в первых четырех байтах) таблицы виртуальных функций. Далее используется следующий механизм:

```
mov  ecx, [ebp+var_4]   ;адрес объекта
mov  edx, [ecx]         ;содержимое начала области, где хранится
;объект, в регистр EDX. Это и есть адрес таблицы виртуальных функций
mov  ecx, [ebp+var_4]
call dword ptr [edx]    ;вызов виртуальной функции
```

Итак, мы видим, что адрес виртуальной функции, в отличие от обычной, формируется динамически. Таким образом, любой косвенный вызов (независимо от компилятора) должен настораживать нас — очень вероятно, что перед нами как раз вызов виртуальной функции. В нашей программе виртуальные функции вызываются дважды: `call dword ptr [edx]` (это `seta`) и `call dword ptr [edx+4]` (это `seta1`). Регистр EDX указывает на начало таблицы виртуальных функций. Всего, очевидно, в таблице должны содержаться адреса двух виртуальных функций.

Перейдем теперь к листингу 3.79, где содержится текст функции `sub_4010A0`, основное назначение которой — занести адрес таблицы виртуальных функций в начало области памяти, где хранится объект.

Листинг 3.79

```
.text:004010A0  sub_4010A0    proc near   ; CODE XREF: _main+1C?p
.text:004010A0    var_4   = dword ptr -4
.text:004010A0      push    ebp
.text:004010A1      mov     ebp, esp
.text:004010A3      push    ecx
.text:004010A4      mov     [ebp+var_4], ecx
.text:004010A7      mov     ecx, [ebp+var_4]
.text:004010AA      call    ??0ios_base@std@@IAE@XZ
.text:004010AF      mov     eax, [ebp+var_4]
.text:004010B2      mov     dword ptr [eax], offset off_407100
.text:004010B8      mov     eax, [ebp+var_4]
.text:004010BB      mov     esp, ebp
.text:004010BD      pop     ebp
.text:004010BE      retn
.text:004010BE  sub_4010A0  endp
```

Комментарий к листингу 3.79

Функция, представленная в листинге, устроена очень интересно. Во-первых, здесь определяется адрес таблицы виртуальных функций и заносится в начало области, где хранится объект. Вот эти команды:

```
...
mov  [ebp+var_4], ecx
...
mov  eax, [ebp+var_4]
mov  dword ptr [eax], offset off_407100
...
```

Да, очевидно в области памяти с адресом `off_407100` и содержится таблица виртуальных функций. А как быть с вызовом функции `??0ios_base@std@@IAE@XZ`. Здесь нужно понять следующее. В нашей программе иерархия классов имеет всего два уровня: базовый класс `A` и производный класс `B`. Таблицы виртуальных функций создаются для каждого класса. Функция `??0ios_base@std@@IAE@XZ` заносит в объект адрес таблицы виртуальных функций базового класса. Разумеется, в нашем случае адрес перезаписывается адресом таблицы виртуальных функций класса `B`. Теперь представьте себе, что в нашу иерархию мы добавляем еще один класс `C`, который является потомком класса `B`, и создаем объект на основе класса `C`: `A1=new(C)`. Как в этом случае компилятор будет обходиться с таблицами виртуальных функций? А вот как (листинг 3.80).

Листинг 3.80

```
;функция main
_main proc near
  ...
  call proc1
;теперь объект содержит адрес таблицы виртуальных функций класса C
  ...
_main endp
...
proc1 proc near
  ...
  call proc2
;теперь объект содержит адрес таблицы виртуальных функций класса B
  ...
;теперь объект содержит адрес таблицы виртуальных функций класса C
  ...
  retn
proc1 endp
        ...
Proc2 proc near
  ...
  call ??0ios_base@std@@IAE@XZ
  ...
;теперь объект содержит адрес таблицы виртуальных функций базового класса
  ...
```

```
;теперь объект содержит адрес таблицы виртуальных функций класса B
  ...
  retn
proc1 endp
```

Внимательно разберите схему, представленную в листинге 3.80. Из нее должно быть понятно, как осуществляется запись адреса виртуальных функций для иерархии объектов с произвольным числом членов. Отсюда, в частности, вытекает, что чем больше членов, тем больше времени потребуется для выполнения такой операции — количество вложенных процедур как раз равно количеству классов, включая базовый класс. Замечу также, что компилятор Visual C++ располагает таблицы виртуальных функций родственных классов друг за другом, причем базовый класс имеет наибольший адрес (адреса растут снизу вверх). Сами же функции в таблице располагаются снизу вверх согласно их объявлению в тексте программы.

Зная адрес таблицы виртуальных функций, мы легко выйдем на текст этой функции. Вот, например, как выглядит код виртуальной функции `seta`, определенной в классе `B` (листинг 3.81).

Листинг 3.81

```
.text:004010C0  sub_4010C0  proc near ;DATA XREF: .rdata:off_407100?o
.text:004010C0    var_4  = dword ptr -4
.text:004010C0    arg_0  = dword ptr  8
.text:004010C0      push    ebp
.text:004010C1      mov     ebp, esp
.text:004010C3      push    ecx
.text:004010C4      mov     [ebp+var_4], ecx
.text:004010C7      mov     eax, [ebp+arg_0]
.text:004010CA      add     eax, 1
.text:004010CD      mov     ecx, [ebp+var_4]
.text:004010D0      mov     [ecx+4], eax
.text:004010D3      mov     edx, [ebp+var_4]
.text:004010D6      mov     eax, [edx+4]
.text:004010D9      mov     esp, ebp
.text:004010DB      pop     ebp
.text:004010DC      retn    4
.text:004010DC  sub_4010C0  endp
```

Комментарий к листингу 3.81

Текст процедуры вполне понятен. Во-первых, в регистре ECX, как и ранее, содержится адрес объекта. Параметр же arg_0 — это как раз то значение, которое должно быть присвоено свойству a (см. листинг 3.77). Наконец, при присвоении значение должно быть еще увеличено на 1:

```
add  eax, 1
mov  ecx, [ebp+var_4]
mov  [ecx+4], eax
```

Мы видим, что свойство a располагается со смещением 4 от начала объекта. И это правильно, поскольку в первых четырех байтах находится адрес таблицы виртуальных функций. При установке опций оптимизации компилятор сокращает код. Например, исчезают процедуры, где в объект записывается адрес виртуальной функции, — все происходит непосредственно в функции main (я имею в виду наш пример). Кроме этого, в объект записывается сразу адрес последнего класса.

При обращении к компилятору Borland C++ 5.0 мы не обнаружим никаких особых отличий. Адрес таблицы виртуальных функций также размещается в начале объекта. Он также при инициализации перезаписывается, начиная с базового класса (точнее с класса, в котором впервые появилось ключевое слово virtual) и заканчивая текущим производным классом.

Конструктор и деструктор

Необходимость *конструктора* и *деструктора* логически вытекает из концепции объектного программирования, особенно применительно к визуальному программированию в операционной системе Windows. Действительно, имеется насущная необходимость автоматизировать действия, которые должны производиться при создании объекта и при его уничтожении. В частности, это касается начальных значений свойств объекта, которые нельзя инициализировать при их задании, как это принято в языке C. Вызывать же каждый раз процедуру инициализации — дурной тон программирования[13]. До сих пор во всех примерах программ я не использовал ни конструкторов, ни деструкторов. Однако этого нельзя сказать о компиляторе. Ведь определение адреса таблицы виртуальных функций — это как раз то самое действие, которое должно производиться автоматически. Так-так, а с конструктором то мы уже встречались! Впервые с конструктором мы встретились в листинге 3.78. Это процедура sub_4010A0, которую мы отдельно разобрали в листинге 3.79. Единственное назначение этой процедуры — записать в начало объекта адрес правильной таблицы виртуальных функций. Но если мы

[13] При создании очередного объекта программист может забыть вызвать процедуру инициализации или же вызвать ее дважды.

определим конструктор явно и выполним там конкретные действия, то эти действия окажутся "в одной компании" с действиями по определению таблицы виртуальных функций.

Перейдем к практическим действиям и рассмотрим следующую программу (листинг 3.82). В программе определен один класс A, в котором имеется одна переменная, один виртуальный метод, а также специальные методы: конструктор и деструктор.

Листинг 3.82

```
#include <stdio.h>
class A {
public:
  int a;
  virtual void pa(){printf("%d\n",a);}
  A(){a=1; printf("Constructor A\n");};
  ~A(){printf("Destructor A\n");};
};
void main()
{
  A* A1;
  A1=new(A);
  A1->pa();
  delete A1;
};
```

В листинге 3.83 вы можете видеть дизассемблированный исполняемый код функции main. Код создан компилятором Visual C++ без оптимизации.

Листинг 3.83

```
.text:00401000  _main    proc near    ; CODE XREF: start+16E?p
.text:00401000    var_18  = dword ptr -18h
.text:00401000    var_14  = dword ptr -14h
.text:00401000    var_10  = dword ptr -10h
.text:00401000    var_C  = dword ptr -0Ch
.text:00401000    var_8  = dword ptr -8
.text:00401000    var_4  = dword ptr -4
.text:00401000      push    ebp
.text:00401001      mov     ebp, esp
```

```
.text:00401003      sub     esp, 18h
.text:00401006      push    8
.text:00401008      call    ??2@YAPAXI@Z     ; operator new(
.text:0040100D      add     esp, 4
.text:00401010      mov     [ebp+var_8], eax
.text:00401013      cmp     [ebp+var_8], 0
.text:00401017      jz      short loc_401026
.text:00401019      mov     ecx, [ebp+var_8]
.text:0040101C      call    sub_401070
.text:00401021      mov     [ebp+var_14], eax
.text:00401024      jmp     short loc_40102D
.text:00401026  loc_401026:        ; CODE XREF: _main+17?j
.text:00401026      mov     [ebp+var_14], 0
.text:0040102D  loc_40102D:        ; CODE XREF: _main+24?j
.text:0040102D      mov     eax, [ebp+var_14]
.text:00401030      mov     [ebp+var_4], eax
.text:00401033      mov     ecx, [ebp+var_4]
.text:00401036      mov     edx, [ecx]
.text:00401038      mov     ecx, [ebp+var_4]
.text:0040103B      call    dword ptr [edx]
.text:0040103D      mov     eax, [ebp+var_4]
.text:00401040      mov     [ebp+var_10], eax
.text:00401043      mov     ecx, [ebp+var_10]
.text:00401046      mov     [ebp+var_C], ecx
.text:00401049      cmp     [ebp+var_C], 0
.text:0040104D      jz      short loc_40105E
.text:0040104F      push    1
.text:00401051      mov     ecx, [ebp+var_C]
.text:00401054      call    sub_4010C0
.text:00401059      mov     [ebp+var_18], eax
.text:0040105C      jmp     short loc_401065
.text:0040105E  loc_40105E:        ; CODE XREF: _main+4D?j
.text:0040105E      mov     [ebp+var_18], 0
.text:00401065  loc_401065:        ; CODE XREF: _main+5C?j
.text:00401065      xor     eax, eax
.text:00401067      mov     esp, ebp
.text:00401069      pop     ebp
.text:0040106A      retn
.text:0040106A  _main    endp
```

Комментарий к листингу 3.83

- Структура листинга нам, конечно же, знакома. Но я хотел обратить ваше внимание на моменты, которые я еще не освещал. Итак, из предыдущего материала очевидно, что функция `sub_401070` — это самый настоящий конструктор. Об этом говорит, во-первых, близость его к оператору `new`. Во-вторых, и это тоже важно, после выполнения оператора `new` осуществляется проверка, выделена в действительности была память или нет. Просмотрите внимательно листинги из *разд. 3.2.4.* Вы увидите, что такая проверка возникает только с появлением виртуальных функций. Но мы-то с вами уже знаем, что при наличии виртуальных функций компилятор создает конструктор, даже если его и не было в исходном тексте программы. И вот это, пожалуй, является главным признаком вызова конструктора, потому что компилятор не желает допустить выполнения конструктора, если объект не был создан.
- Перейдем теперь к концу функции. Обращает на себя внимание вызов процедуры `sub_4010C0`. Ранее в подобных программах там стоял вызов функции `j__free`, который мы ассоциировали с оператором `delete` (см. листинг 3.74 и комментарий к нему). Очевидно перед нами самый натуральный деструктор, в чем мы убедимся в листинге 3.85. Замечу, что и тут компилятор не позволяет выполниться деструктору, если объект не был создан.
- Что касается остального кода, то я надеюсь, что читатель сам легко опознает (после многократного обсуждения подобных фрагментов в предыдущих листингах) в команде `call dword ptr [edx]` вызов виртуальной функции
  `A1->pa()`.

Посмотрим теперь, что у нас помещено в конструктор (листинг 3.84).

Листинг 3.84

```
.text:00401070  sub_401070  proc near     ; CODE XREF: _main+1C?p
.text:00401070    var_4  = dword ptr -4
.text:00401070      push   ebp
.text:00401071      mov    ebp, esp
.text:00401073      push   ecx
.text:00401074      mov    [ebp+var_4], ecx
.text:00401077      mov    eax, [ebp+var_4]
.text:0040107A      mov    dword ptr [eax], offset off_40710C
.text:00401080      mov    ecx, [ebp+var_4]
.text:00401083      mov     dword ptr [ecx+4], 1
```

```
.text:0040108A      push offset aConstructorA ; "Constructor A\n"
.text:0040108F      call   _printf
.text:00401094      add    esp, 4
.text:00401097      mov    eax, [ebp+var_4]
.text:0040109A      mov    esp, ebp
.text:0040109C      pop    ebp
.text:0040109D      retn
.text:0040109D  sub_401070  endp
```

Комментарий к листингу 3.84

Листинг, несомненно, должен нас обрадовать. Посмотрите, перед нами уже знакомое:

```
mov  [ebp+var_4], ecx
mov  eax, [ebp+var_4]
mov  dword ptr [eax], offset off_40710C
```

Это же не что иное, как занесение адреса таблицы виртуальных функций в созданный экземпляр объекта. А далее:

```
;задать значение свойства  (a=1)
mov     dword ptr [ecx+4], 1
;вызов функции printf
push    offset aConstructorA ; "Constructor A\n"
call    _printf
```

Но это же наш текст, который мы сами поместили в конструктор (см. листинг 3.82). Собственно, мне больше нечего добавить — вся картина абсолютно ясна.

В листинге 3.85 представлен дизассемблированный текст деструктора (точнее, той процедуры, из которой истинный деструктор и будет вызван) для программы из листинга 3.82.

Листинг 3.85

```
.text:004010C0  sub_4010C0    proc near  ; CODE XREF: _main+54?p
.text:004010C0    var_4  = dword ptr -4
.text:004010C0    arg_0  = dword ptr  8
.text:004010C0      push   ebp
.text:004010C1      mov    ebp, esp
.text:004010C3      push   ecx
.text:004010C4      mov    [ebp+var_4], ecx
```

```
.text:004010C7      mov    ecx, [ebp+var_4]
.text:004010CA      call   sub_4010F0
.text:004010CF      mov    eax, [ebp+arg_0]
.text:004010D2      and    eax, 1
.text:004010D5      jz     short loc_4010E3
.text:004010D7      mov    ecx, [ebp+var_4]
.text:004010DA      push   ecx
.text:004010DB      call   j__free
.text:004010E0      add    esp, 4
.text:004010E3  loc_4010E3:         ; CODE XREF: sub_4010C0+15?j
.text:004010E3      mov    eax, [ebp+var_4]
.text:004010E6      mov    esp, ebp
.text:004010E8      pop    ebp
.text:004010E9      retn   4
.text:004010E9  sub_4010C0  endp
```

Комментарий к листингу 3.85

В листинге заметим в первую очередь два вызова: sub_4010F0 и j__free. Заглянув в sub_4010F0, мы убедимся, что там содержится просто текст, который мы поместили в деструктор (см. листинг 3.82). Таким образом, функция sub_4010C0 предназначена для вызова процедур, необходимых при уничтожении объекта. Функцию j__free мы уже знаем — это просто реализация оператора delete. Наконец, единица, которая посылается в функцию sub_4010C0 в качестве параметра, является признаком необходимости вызова оператора delete.

3.2.5. Еще об исследовании исполняемого кода

О математических вычислениях

Мы уже встречались с различными математическими вычислениями. Часто, для того чтобы сократить код или ускорить работу программы, компиляторы применяют различные приемы. С таким приемом, как вычисление на стадии компиляции, мы уже знакомились. Знаем мы также и возможности использования команд арифметического сопроцессора. О некоторых других приемах мы поговорим в данном разделе.

В листинге 3.86 представлена программа, содержащая простое арифметическое вычисление. Посмотрим, как решает поставленную задачу компилятор Visual C++ при условии, что мы задаем опцию "создавать быстрый код". Дизассемблированный текст исполняемого кода представлен в листинге 3.87.

Листинг 3.86

```
#include <stdio.h>
#include <windows.h>
void main()
{
  DWORD a,b,c;
  scanf("%d",&a);
  scanf("%d",&b);
  c=((a+b)/8)*(3*a);
  printf("%d\n",c);
};
```

Листинг 3.87

```
.text:00401000  _main    proc near    ; CODE XREF: start+16E?p
.text:00401000    var_8  = dword ptr -8
.text:00401000    var_4  = dword ptr -4
.text:00401000      sub    esp, 8
.text:00401003      lea    eax, [esp+8+var_8]
.text:00401006      push   eax
.text:00401007      push   offset unk_408100
.text:0040100C      call   _scanf
.text:00401011      lea    ecx, [esp+10h+var_4]
.text:00401015      push   ecx
.text:00401016      push   offset unk_408100
.text:0040101B      call   _scanf
.text:00401020      mov    ecx, [esp+18h+var_8]
.text:00401024      mov    edx, [esp+18h+var_4]
.text:00401028      lea    eax, [edx+ecx]
.text:0040102B      shr    eax, 3
.text:0040102E      imul   eax, ecx
.text:00401031      lea    eax, [eax+eax*2]
.text:00401034      push   eax
.text:00401035      push   offset unk_4080FC
.text:0040103A      call   _printf
.text:0040103F      xor    eax, eax
.text:00401041      add    esp, 20h
.text:00401044      retn
.text:00401044  _main    endp
```

Комментарий к листингу 3.87

Обратим внимание, что адресация стековых переменных в данном фрагменте производится посредством регистра ESP. Итак, стековые переменные var_4 и var_8 — это переменные a и b (см. листинг 3.86). Рассмотрим последовательность команд:

```
mov  ecx, [esp+18h+var_8]
mov  edx, [esp+18h+var_4]
lea  eax, [edx+ecx]
```

Главное здесь — использование команды lea. О необычных свойствах этой команды мы уже говорили (см. табл. 1.2). Сейчас мы видим эту команду в действии. Вы часто будете встречаться с этой командой в тех случаях, когда необходимо оптимизировать арифметические действия.

Далее — команда shr eax,3. Конечно, для нас не тайна, что это простое целочисленное деление на 8 (2 в степени 3). Данная команда выполняется много быстрее, чем IDIV. Далее imul eax,ecx — умножить содержимое EAX на содержимое ECX. И, наконец, опять команда lea: lea eax,[eax+eax*2], что означает просто умножение содержимого EAX на 3.

На этом заканчиваем обсуждение математических вычислений. С другими способами оптимизировать вычисления, в частности, с использованием битовых команд, я надеюсь, вы легко справитесь при самостоятельном исследовании исполняемого кода.

Другие конструкции

Обработка исключений

Механизмы обработки исключений на уровне исполняемого кода довольно сложны и в деталях могут значительно отличаться для различных компиляторов. Но в мою задачу и не входит изучение этих механизмов. Я ставлю перед собой куда более скромные цели: познакомиться с некоторыми базовыми принципами обработки исключений и тем, как это может проявляться в исполняемом коде.

Обработка исключений, генерируемая компиляторами, основывается на так называемой *структурной обработке исключений* (Structured Exception Handling, SEH). Механизм SEH поддерживается на уровне операционной системы Windows. Итак, в основе обработки исключений лежат следующие факты. В операционных системах Windows на основе процессора Intel сегментный регистр FS играет особую роль. Он указывает на *блок окружения потока*[14]. Он называется еще TEB (Thread Environment Block). В свою оче-

[14] Напомню, что в защищенном режиме сегментные регистры хранят не адреса, а селекторы. Селектор — это фактически номер в таблице дескрипторов, которые и определяют адрес.

редь, одной из подструктур, которая находится в начале этого блока, является TIB (Thread Information Block) — информационный блок потока. Наконец, четырехбайтовая величина, находящаяся в самом начале структуры TIB, является адресом структуры, которая, во-первых, содержит адрес обработчика исключений, а во-вторых, адрес предыдущей подобной структуры. В действительности речь идет о связанном списке, но об этом подробнее мы поговорим ниже (см. комментарий к листингу 3.89).

Какие выводы можно сделать из сказанного:

- каждый поток имеет свой обработчик исключений;
- зная адрес, где находится указатель на обработчик исключения, программа сама может установить свою процедуру обработки.

По сути, эти два факта и положены в основу обработки исключений, которые используют компиляторы, создающие исполняемый код для Windows. И хотя факты просты, сам механизм реализации исключений, как я уже говорил, может быть достаточно сложным.

Когда я говорю о реализации компиляторами механизма исключений, то в первую очередь имею в виду пару таких операторов, реализованных в C++[15], как `__try...__except`. Рассмотрим программу из листинга 3.88. Программа очень проста. Блок `__try` используется для того, чтобы обезопасить нас от возможного деления на 0 при вычислении частного от деления `a` на `b`.

Листинг 3.88

```
#include <stdio.h>
#include <windows.h>

int main()
{
  int a,b;
  scanf("%d",&a);
  scanf("%d",&b);
  __try  {
    a=a/b;
    printf("%d\n",a);
  }
  __except(0)
```

[15] Это стандарт для языка C++, поэтому все компиляторы C++ поддерживают эти операторы, хотя реализация может отличаться друг от друга.

```
  {
    printf("Error 1! \n");
  };
  return 0;
}
```

В листинге 3.89 представлен исполняемый код, предварительно дизассемблированный программой IDA Pro. Программа была откомпилирована при помощи компилятора Visual C++ с установленной опцией "создавать быстрый код".

Листинг 3.89

```
.text:00401000  _main    proc near    ; CODE XREF: start+16E?p
.text:00401000    var_20  = dword ptr -20h
.text:00401000    var_1C  = dword ptr -1Ch
.text:00401000    var_18  = dword ptr -18h
.text:00401000    var_10  = dword ptr -10h
.text:00401000    var_4  = dword ptr -4
.text:00401000      push    ebp
.text:00401001      mov     ebp, esp
.text:00401003      push    0FFFFFFFFh
.text:00401005      push    offset byte_408110
.text:0040100A      push    offset __except_handler3
.text:0040100F      mov     eax, large fs:0
.text:00401015      push    eax
.text:00401016      mov     large fs:0, esp
.text:0040101D      sub     esp, 10h
.text:00401020      push    ebx
.text:00401021      push    esi
.text:00401022      push    edi
.text:00401023      mov     [ebp+var_18], esp
.text:00401026      lea     eax, [ebp+var_1C]
.text:00401029      push    eax
.text:0040102A      push    offset aD_0     ; "%d"
.text:0040102F      call    _scanf
.text:00401034      lea     ecx, [ebp+var_20]
.text:00401037      push    ecx
.text:00401038      push    offset aD_0     ; "%d"
```

```
.text:0040103D     call    _scanf
.text:00401042     add     esp, 10h
.text:00401045     mov     [ebp+var_4], 0
.text:0040104C     mov     eax, [ebp+var_1C]
.text:0040104F     cdq
.text:00401050     idiv    [ebp+var_20]
.text:00401053     mov     [ebp+var_1C], eax
.text:00401056     push    eax
.text:00401057     push    offset aD       ; "%d\n"
.text:0040105C     call    _printf
.text:00401061     add     esp, 8
.text:00401064     jmp     short loc_401079
.text:00401066 ;---------------------------------------------------------------
.text:00401066     xor     eax, eax
.text:00401068     retn
.text:00401069 ;---------------------------------------------------------------
.text:00401069     mov     esp, [ebp-18h]
.text:0040106C     push    offset aError1  ; "Error 1! \n"
.text:00401071     call    _printf
.text:00401076     add     esp, 4
.text:00401079 loc_401079:          ; CODE XREF: _main+64?j
.text:00401079     mov     [ebp+var_4], 0FFFFFFFFh
.text:00401080     xor     eax, eax
.text:00401082     mov     ecx, [ebp+var_10]
.text:00401085     mov     large fs:0, ecx
.text:0040108C     pop     edi
.text:0040108D     pop     esi
.text:0040108E     pop     ebx
.text:0040108F     mov     esp, ebp
.text:00401091     pop     ebp
.text:00401092     retn
.text:00401092 _main    endp
```

Комментарий к листингу 3.89

- Обратите внимание на наличие стандартного пролога функции, хотя в обычных случаях при оптимизированной компиляции стандартные прологи и эпилоги опускаются. Обратите также внимание, что IDA Pro указывает аж пять стековых переменных, хотя в исходном коде программы используются всего две. Бегло просмотрев код, мы легко определяем (например, по вызову функций `scanf`), что `var_1C` соответствует в тексте

переменной a, а var_20 — переменной b. Хотя в тексте программы мы можем ограничивать с помощью блока __try лишь часть кода функции, но, по сути, этот блок воздействует (преобразует) на всю функцию.

- Следом за прологом идет интересная последовательность. Приведу эти команды еще раз и прокомментирую.

```
push  0FFFFFFFFh                ;var_4
push  offset byte_408110        ;
push  offset __except_handler3  ;адрес нового обработчика исключений
mov   eax, large fs:0           ;адрес старой структуры обработки
push  eax                       ;var_10
mov   large fs:0, esp           ;адрес нового обработчика
```

  Вначале в стек кладется константа, значение которой произвольно. Ниже это значение будет изменено. В действительности это значение в дальнейшем должно определять уровень вложенности блока __try. Затем в стек помещается адрес некоторой глобальной ячейки памяти. Третьим идет адрес обработчика исключений, сформированный компилятором. Последние три команды очень интересны. В стек помещается адрес старой структуры обработки исключений, а затем адрес всей группы из четырех двойных слов (это новая структура обработки исключений) в стеке помещается в fs:0. Данная операция есть не что иное, как добавление новой записи к связанному списку. Этот список может содержать целую цепочку обработчиков исключений. Последняя структура в списке должна содержать в адресе обработчика исключений значение 0FFFFFFFFH. Замечу также, что в стеке находится и старое значение EBP.

- Обратим внимание на инструкцию mov [ebp+var_4],0. Она стоит перед самой командой выполнения действия деления. Она задает уровень вложенности блока __try. Замечу, что вложенность определяется количеством блоков __try в функции в независимости от того, находится один блок внутри другого или нет. Следите за содержимым var_4, т. к. это ключ к структуре __try блоков функции.

- Значение регистра ESP сохраняется на момент полного формирования стека функции: mov [ebp+var_18],esp. Это значение используется для восстановления стека в блоке __except (см. ниже).

- Не останавливаясь на стандартных ассемблерных командах и вызове функций scanf и printf, перейдем к блоку __except. Найти начало и конец этого блока достаточно легко, т. к. перед ним располагаются неизменные команды

```
jmp    short loc_401079
xor    eax, eax
retn
```

Эти команды будут во всех вариантах оптимизации компилятора Visual C++. Конец `__except` блока определяется командой `jmp`.

Если обратиться к компилятору Borland C++ 5.0, то, несмотря на то, что здесь реализован несколько иной механизм обработки исключений, внешние признаки исключений довольно прозрачны и легко распознаются. Во-первых, это наличие в начале процедуры блока установки исключения (замена значения, хранящегося по адресу `FS:0`). Во-вторых, наличие в центре процедуры фрагмента, на который отсутствует переход и который обходится командой `JMP`. Наконец, наличие в конце процедуры команды восстановления старого значения `FS:0`.

Идентификация главной функции и начальный код

Ранее мы обращались к главной функции — к функции, с которой начинается программа, так, как будто узнать ее адрес не составляет никакого труда. Да, действительно, адрес функции `main` в исполняемом коде программы, скомпилированной с помощью компилятора C++, IDA Pro определяет без труда. Но, во-первых, IDA Pro не всегда под рукой, а во-вторых, есть и другие языки программирования.

В программе на любом алгоритмическом языке программирования есть некоторый начальный набор команд. Повторюсь, для C++ это функция `main` или `WinMain`. Однако не надо считать, что в исполняемом коде именно с этого набора команд и начинается выполняться программа. Как правило, перед этим компилятор вставляет некоторый стартовый код, содержащий вызовы библиотечных и API-функций. Этот код выполняет некоторую начальную подготовку: запрашивает у системы память, определяет параметры командной строки, получает идентификатор исполняемого модуля и др. И только после этого управление передается тем начальным командам, о которых мы говорили. Как я уже отмечал, дизассемблеры (и отладчики) не всегда могут определить, где начинаются эти команды.

Замечание

Если говорить о компиляторе Visual C++, то исполняемый код в случае консольной программы начинает выполняться со стартовой функции `mainCRTStartup`, которая затем передает управление функции `main`. Для приложений GUI выполнение начинается со стартовой функции `WinMainCRTStartup`, которая опять же в конце передает управление функции `WinMain`. Существует методика, позволяющая создавать исполняемые модули, которые начинают исполняться сразу с функции `main` (`WinMain`). Это приводит к значительному уменьшению размера исполняемого модуля (на размер стандартных библиотек). Однако такая практика случается совсем не часто, т. к. в этом случае программист лишает себя возможности использовать стандартные библиотеки языка С.

Итак, чтобы искать начало пользовательской части исполняемого кода, нужно знать, как происходит этот вызов. Начнем с консольного приложения на языке C++. Прообраз функции `main` в общем случае имеет вид:

```
int main( int argc[ , char *argv[ ] [, char *envp[ ] ] ] );
```

То есть функция имеет в общем случае три входных параметра. Это важный признак. Вот типичный пример вызова функции `main` из исполняемого кода, созданного компилятором Visual C++:

```
mov     eax, dword_40A724
mov     dword_40A728, eax
push    eax
push    dword_40A71C
push    dword_40A718
call    _main
add     esp, 0Ch
```

Обратите внимание, что все три параметра оказались глобальными переменными. Это существенный момент. Очень важно "осмотреться" вокруг и поизучать, какие библиотечные и API-функции вызываются перед и после вызова функции `main`. Зная это, вы легко сможете найти нужное место. Особо отмечу вызов функции API `GetCommandLine`. С ее помощью можно получить параметры командной строки. Именно эти параметры затем передаются во втором параметре функции `main`. Вы можете спросить меня о том, что произойдет, если функция `main` не будет иметь параметров или параметров окажется два или один? Ничего не произойдет, и вызов будет происходить точно так же. Так что критерии поиска никак не меняются.

В случае с Borland C++ все гораздо сложнее. Вход в функцию `main` осуществляется косвенным вызовом вида `CALL [ESI+N]`. В том случае, если дизассемблер не определяет функцию `main` автоматически, вам придется использовать отладчик. Но замечания по поводу предварительного вызова библиотечных и API-функций (опять же `GetCommandLine`) вполне годятся и в этом случае. Зная эти функции, вы сможете выйти на нужный участок, а затем на вызов типа `CALL [ESI+N]`. Здесь опять придется использовать отладчик и точки останова, т. к. ручной анализ того, по какому адресу осуществляется вызов, утомителен.

Рассмотрим теперь вызов функции `WinMain`. Вот прообраз этой функции:

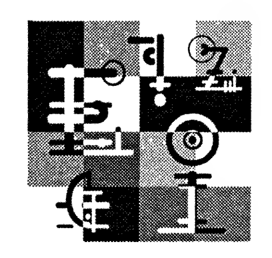
```
int WinMain(HINSTANCE hInstance,
            HINSTANCE hPrevInstance,
            LPSTR lpCmdLine,
            int nCmdShow
);
```

Как видим, функция имеет аж четыре параметра. И это также является важным признаком поиска начала программы. Вот типичный фрагмент вызова главной функции WinMain из исполняемого модуля, созданного компилятором Visual C++:

```
push    eax
push    dword ptr [ebp-20h]
push    esi
push    esi              ; lpModuleName
call    edi              ; GetModuleHandleA
push    eax
call    _WinMain@16      ; WinMain(x,x,x,x)
mov     edi, eax
mov     [ebp-2Ch], edi
cmp     [ebp-1Ch], esi
jnz     short loc_401508
push    edi              ; int
call    _exit
```

Фрагмент, который мы видим, настолько характерен, что стоит его запомнить. Он дает безошибочный критерий поиска входа в начальный код исполняемой программы. Что касается вызова функции WinMain в модуле, созданном компилятором Borland C++, то здесь опять вызов осуществляется посредством команды CALL [ESI+N]. Причем перед этим вызывается API-функция GetModuleHandle.

В Delphi исполнение начинается с главного модуля программы (begin...end.), но перед этим стоят вызовы одной или нескольких процедур, осуществляющих начальную инициализацию. В частности, в одной из этих процедур вызывается все та же API-функция GetModuleHandle.



Глава 4

Отладчик SoftICE

В настоящее время отладчик SoftICE существует для всех операционных систем семейства Windows и даже MS-DOS. Прежде всего, замечу, что SoftICE — это отладчик уровня ядра (kernel mode debugger). С его помощью можно отлаживать любые программы, исполняемые в операционной системе, в том числе сервисы и драйверы, работающие в нулевом кольце защиты. По причине тесного взаимодействия отладчика с операционной системой с его помощью можно получить много системной (я бы сказал, довольно личной) информации о функционировании операционных систем. Поэтому SoftICE просто незаменим для тех, кто изучает внутренние механизмы функционирования операционной системы Windows. В среде исследователей кода SoftICE считается лучшим[1] отладчиком.

Сам отладчик дополняется также утилитами, главная из которых — это Symbol Loader (загрузчик отладочной информации). Программа Symbol Loader (loader32.exe) загружает исполняемый модуль в память и осуществляет вызов окна отладчика SoftICE, другими словами, устанавливает точку останова на точку входа программы. При наличии в исполняемом модуле отладочной информации, распознаваемой загрузчиком, он также загружает ее в отладчик. Отладчик позволяет отлаживать исполняемый код не только на автономном компьютере, но и производить удаленную отладку (remote debugger) — с удаленного компьютера, соединенного с первым посредством COM-порта.

Установка SoftICE — это отдельная статья. Поскольку отладчик работает на уровне ядра, то разработчикам постоянно приходится дорабатывать свой продукт, дабы его можно было использовать со всеми релизами операционных систем Windows. И, тем не менее, Интернет полон статей и обсуждений, посвященных проблемам установки SoftICE и устранению различных

---

[1] Название отладчика SoftICE — это указание на то, что отладчик в любой момент может "заморозить" (от англ. *to ice* — замораживать) систему и дать полную информацию и по системе, и по всем работающим в ней приложениям.

проблем во время работы этого продукта. Чтобы не загромождать книгу, я опущу вопросы установки отладчика и отошлю заинтересованного читателя к сайту поддержки **http://www.compuware.com**, где, в частности, после регистрации вы можете получить руководства по SoftICE (Reference Guide). Отмечу также русскоязычный сайт **http://www.wasm.ru**, где можно найти материалы и обсуждения, касающиеся различных технических проблем, возникающих при установке SoftICE.

Моя цель, которую я преследую в данной книге, — дать вводное руководство по отладке приложений при помощи SoftICE. По этой причине я настроен довольно полно описать команды SoftICE, чаще всего используемые при отладке обычных приложений, а также привести примеры отладки при наличии в исполняемом модуле отладочной информации и в отсутствие последней.

В настоящей главе все примеры и описания рассматривались мной по отношению к операционным системам Windows XP и Windows Server 2003.

4.1. Основы работы с SoftICE

В данном разделе я предоставлю читателю основные сведения, которые помогут ему начать работу с этим мощным инструментом.

4.1.1. Запуск и интерфейс

Главное окно SoftICE

При появлении окна SoftICE все системные функции оказываются "замороженными", поэтому для того чтобы изобразить окно SoftICE, мне пришлось поставить рядом два компьютера и срисовать его внешний вид (рис. 4.1).

Окно отладчика SoftICE, которое мы будем называть еще *главным окном*, может появляться в четырех случаях:

- по нажатию комбинации клавиш <Ctrl>+<D>. Данное действие приведет к вызову окна отладчика в любом случае при выполнении произвольной программы. Вы, таким образом, можете в любой момент посмотреть состояние системы и исполняемых приложений;
- при загрузке какого-либо приложения в память с помощью программы Loader32.exe. При этом происходит прерывание процесса загрузки как раз на точке входа в исполняемый модуль, и вы можете продолжить выполнение приложения в каком-либо из режимов отладчика с самого начала;
- при выполнении условия одной из точек останова, которую вы установили ранее в окне отладчика. Отладчик покажет при этом то самое место,

которое стало причиной прерывания. Большим преимуществом отладчика SoftICE является возможность одновременной работы с несколькими приложениями. Вы можете устанавливать точки останова сразу для нескольких приложений;

- кроме этого, окно SoftICE может появляться при возникновении системной ошибки или крахе системы (синий экран).

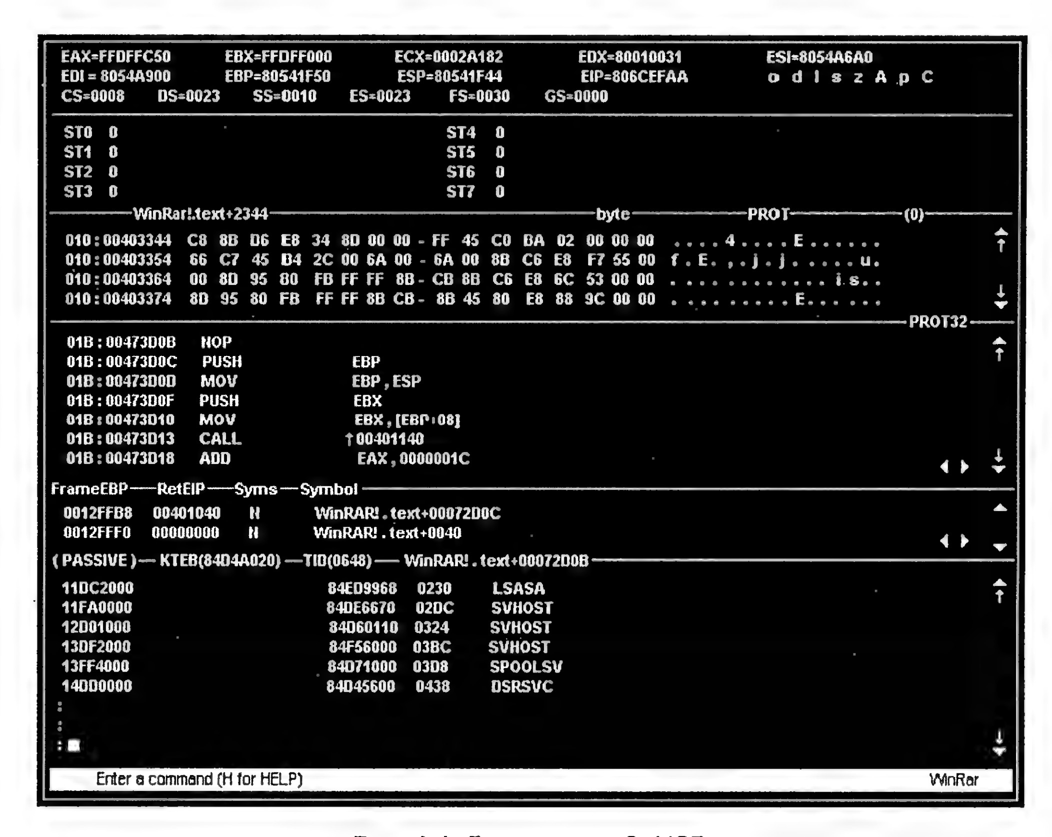
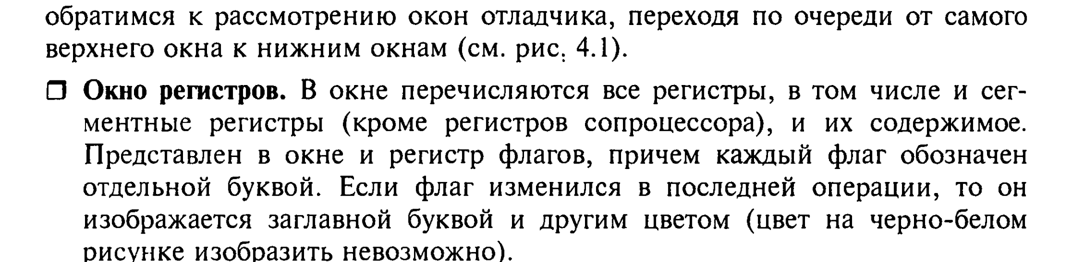


Рис. 4.1. Главное окно SoftICE

Итак, окно отладчика SoftICE представлено на рис. 4.1. В главном окне располагаются еще несколько окон, которые содержат различную информацию. Количество таких окон может быть различно. Вы по своему усмотрению можете добавлять или удалять эти окна в главное окно. На нашем рисунке представлены наиболее востребованные окна отладчика. Особо отмечу, что вы можете не только просматривать информацию в этих окнах, но и менять их содержимое, например, содержимое регистров процессора. Однако делать это следует с большой осторожностью, т. к. это может привести к непредсказуемому поведению приложения и, самое главное, всей системы. Итак,



обратимся к рассмотрению окон отладчика, переходя по очереди от самого верхнего окна к нижним окнам (см. рис. 4.1).

- **Окно регистров.** В окне перечисляются все регистры, в том числе и сегментные регистры (кроме регистров сопроцессора), и их содержимое. Представлен в окне и регистр флагов, причем каждый флаг обозначен отдельной буквой. Если флаг изменился в последней операции, то он изображается заглавной буквой и другим цветом (цвет на черно-белом рисунке изобразить невозможно).
- **Окно регистров сопроцессора.** В окне представлено содержимое всех восьми регистров сопроцессора.
- **Окно данных.** Окно предназначено для представления содержимого области памяти, как в байтовом, так и в ASCII-виде. Можно прокручивать содержимое окна, просматривая, таким образом, произвольные области памяти.
- **Окно кода.** Здесь представлен дизассемблированный код, который также может прокручиваться в окне. Если загружаемое вами приложение содержит отладочную информацию, распознаваемую SoftICE, то в этом окне вы увидите и текст программы на языке высокого уровня.
- **Окно стека.** В окне стека представлено не все содержимое стека, а только стековый кадр, связанный непосредственно с работой данного приложения.
- **Командное окно.** В окне можно набирать многочисленные команды отладчика SoftICE. В частности, из рисунка мы видим, что, набрав команду `H`, можно получить помощь: список команд отладчика. Для получения информации по конкретной команде следует набрать `H` и имя команды, например, так: `H hwnd`.

При работе в главном (командном) окне для управления отладчиком можно использовать команды, которые, как я уже сказал, можно набирать в командном окне, и специальные управляющие сочетания клавиш. Предусмотрено также использование стандартной мыши и контекстного меню.

Обратите также внимание на самое нижнее окно — это окно или панель подсказки. При наборе в командной строке какой-либо команды окно подсказки поможет вам правильно набрать эту команду и ее параметры. В частности, будут перечислены все команды, которые начинаются с тех символов, которые вы уже набрали. Кроме этого, в правом углу окна отладчик всегда показывает текущий процесс. На этот важный момент всегда обращайте внимание, чтобы не перепутать приложения. Мы вернемся к этому вопросу в *разд. 4.1.3*.

Кроме перечисленных выше окон, можно использовать также *окно слежения* — в нем отслеживаются значения переменных, которые указаны в команде `WHATCH`, *окно регистров MMX*, *окно локальных переменных*.

Режимы работы отладчика

После установки отладчика SoftICE вы можете выбрать пять способов запуска:

- **Disable** — отладчик не запускается;
- **Manual** — отладчик не запускается автоматически. Для запуска следует применить команду `Net start ntice`. В каталоге, куда инсталлируется SoftICE, есть пакетный файл ntice.bat с этой командой. Данный режим наиболее безопасен, но в нем невозможно выполнять отладку драйверов устройств на этапе загрузки;
- **Automatic** — отладчик запускается автоматически. В этом режиме, однако, нельзя отлаживать драйверы режима ядра;
- режимы **System** и **Boot** — в обоих случаях отладчик запускается автоматически. Отличие режимов друг от друга заключается в порядке загрузки системных и загрузочных драйверов.

4.1.2. Загрузчик (Loader)

На рис. 4.2 представлено главное окно программы Loader32.exe, предназначенной для загрузки в отладчик исполняемых модулей. Данная утилита умеет также извлекать из отлаживаемых модулей отладочную информацию, если она имеется, и передавать ее отладчику SoftICE. Загружая отлаживаемый модуль, данная утилита устанавливает точку останова на вход в программу.

Загрузка исполняемого модуля

Для того чтобы загрузить в отладчик исполняемый модуль, следует:

1. Открыть его при помощи пункта меню **File | Open**. Можно для этой цели воспользоваться кнопкой **Open** на панели инструментов.
2. Выбрать пункт меню **Module | Load**. Можно также воспользоваться кнопкой **Load Symbols** на панели инструментов. При этом загрузчик вначале транслирует найденную им символьную информацию в файл с расширением nms и таким же именем, как имя программы, а после загружает исходный исполняемый модуль вместе отладочной информацией в отладчик SoftICE. В случае если отладочная информация отсутствует, загрузчик сообщит об этом и предложит выбрать: загружать или нет исследуемый модуль в отладчик. Трансляцию отладочной информации (т. е. создание файла с расширением nms) можно осуществить и отдельной командой: при помощи пункта меню **Module | Translate** или кнопки **Translate** на панели инструментов.

Список **Loaded Symbols** содержит названия загруженных модулей. Обратите внимание на столбец **SYM=**. При загрузке исполняемого модуля здесь будет

содержаться объем загруженной символьной информации. Модули, не содержащие отладочной символьной информации, в список **Loaded Symbols** не заносятся.

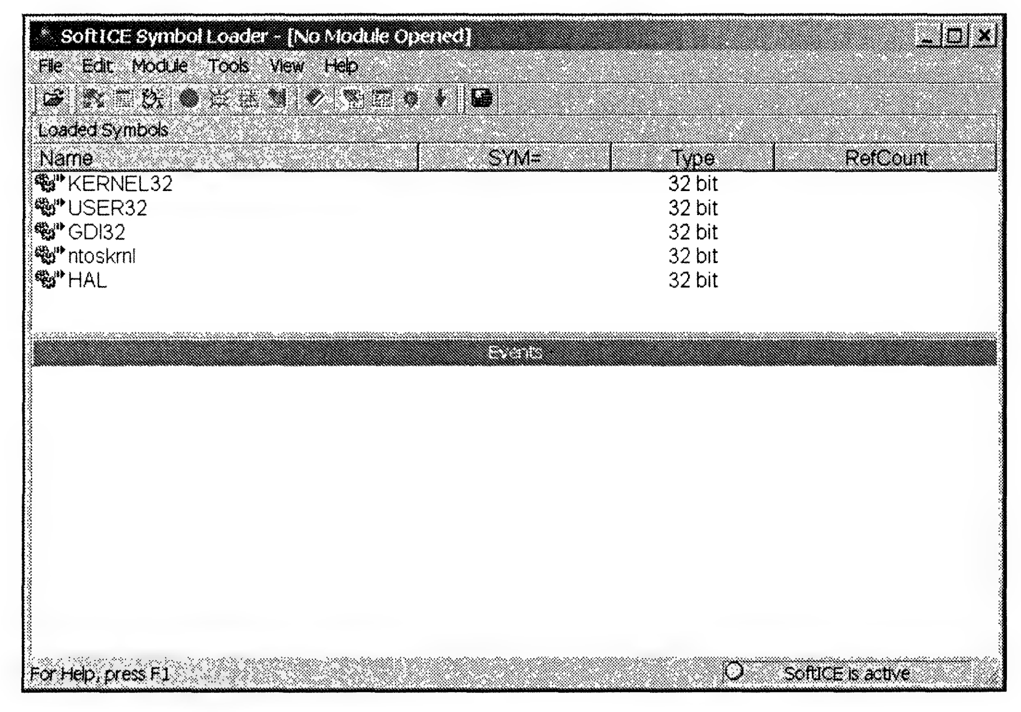


Рис. 4.2. Программа-загрузчик Loader32.exe

Параметры загрузки

После того как модуль, предназначенный для исследования в отладчике SoftICE, открыт, можно установить параметры загрузки. Для этого используется пункт меню **Module | Settings**. Окно, где можно установить параметры загрузки, изображено на рис. 4.3. Оно содержит четыре вкладки. Разберем их подробнее.

- Вкладка **General**:
  - поле редактирования **Command line arguments** — здесь можно задать параметры командной строки, с которыми будет запускаться в отладчике исследуемая программа;
  - поле редактирования **Source file search path** — здесь указываются пути поиска файлов, которые связаны с отлаживаемым модулем;

- поле редактирования **Default source file search path** — здесь задается основной путь для поиска файлов. Отладчик всегда вначале ищет файлы согласно полю **Source file search path** и только потом использует данное поле;
- если флажок **Prompt for missing source files** установлен, то загрузчик при необходимости сообщит вам, что не все файлы, нужные для отладки исполняемого модуля, могут быть загружены. В частности, если отсутствует отладочная информация, вам будет предложено продолжать или не продолжать выполнять загрузку исполняемого модуля в отладчик;
- флажок **Minimize Loader on successful load** используется для минимизации загрузчика в памяти после загрузки исполняемой программы в отладчик.

❒ Вкладка **Debugging**. Позволяет менять некоторые текущие параметры отладки:

- переключатели **Load symbol information only** и **Load executable** позволяют загружать в отладчик только отладочную информацию либо и отладочную информацию, и исполняемый модуль соответственно;
- флажок **Stop at WinMain, main, DllMain etc.** позволяет устанавливать точку прерывания в начало пользовательской части исполняемого модуля. В отсутствие отладочной информации точка прерывания устанавливается в начало выполнения программы.

❒ С помощью вкладки **Translation** задаются параметры трансляции отладочной информации исполняемого модуля:

- переключатель **Publics only** — транслировать только внешние имена;
- переключатель **Type information only** — транслировать только информацию о типах переменных;
- переключатель **Symbols only** — транслировать только символьные имена;
- переключатель **Symbols and source code** — транслировать всю отладочную информацию;
- переключатель **Package source with symbol table** — сохранять оттранслированную информацию в файле формата NMS.

❒ Вкладка **Modules and Files**. Здесь можно перечислить все файлы и их местоположение, которые будут загружены вместе с загружаемым исполняемым модулем. Вы можете перечислить здесь все файлы, содержащие отладочную информацию. Специальным переключателем можно временно заблокировать загрузку того или иного файла.

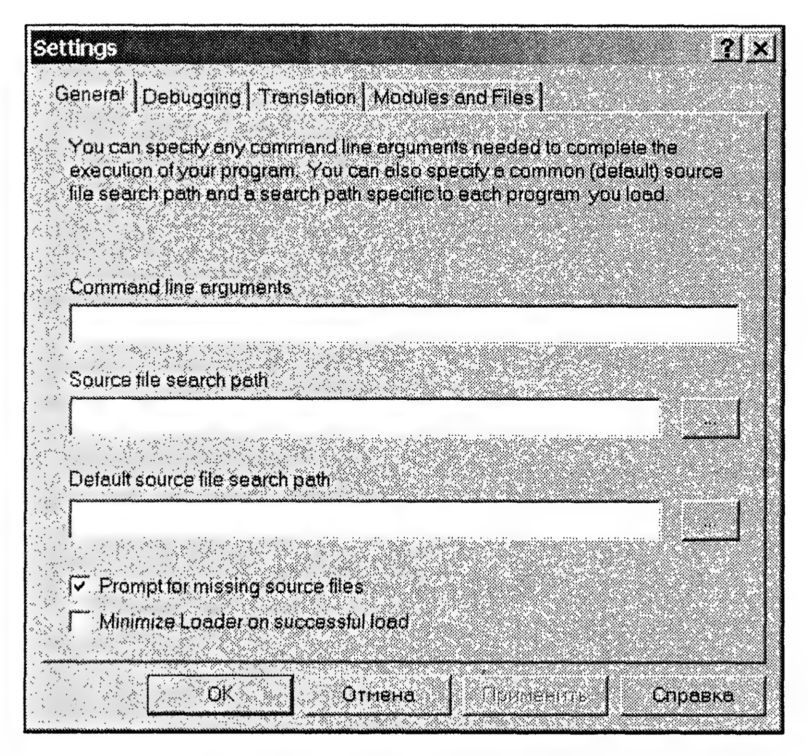


Рис. 4.3. Окно установки параметров загрузки отлаживаемых модулей

4.1.3. Некоторые приемы работы с SoftICE

Начало работы. Процессы

Рассмотрим основные моменты работы с SoftICE.

Мы работаем в многозадачной операционной системе. Программа, которую вы хотите исследовать при помощи SoftICE, после загрузки станет всего лишь одним из многих процессов. Вы должны четко знать, с каким процессом работаете. Не перепутайте процессы, это может привести к зависанию всей системы. Отладчик показывает текущий процесс в правом углу окна подсказки.

При загрузке приложения при помощи программы Loader32.exe остановка происходит на начале выполнения программы. При этом созданный процесс оказывается текущим. Так что вы можете спокойно трассировать загруженное приложение *(см. разд. 4.2.2, команды трассировки)*. Однако когда вы закроете окно отладчика (клавиша <F5>) и снова вызовете его, то данный процесс уже не будет текущим. Каждый запущенный процесс имеет

свое виртуальное адресное пространство *(контекст процесса)*. Все команды, так или иначе связанные с адресами, относятся к конкретному адресному пространству. Например, команда `D DS:004080AF` выдаст содержимое памяти для конкретного виртуального адресного пространства — для текущего процесса. Для того чтобы работать с адресами конкретного процесса, следует сделать процесс текущим. Для этого используйте команду `ADDR` (описание команды *см. в разд. 4.2.2*, информационные команды):

```
: ADDR 058
```

Здесь `058` — это идентификатор процесса (Process Identifier, PID), значение его можно узнать, если использовать команду `ADDR` без параметров.

Основным средством исследования исполняемого кода являются точки останова. Надо четко понимать, куда вы ставите точку останова, т. е. к какому процессу (потоку) относится данная точка останова. В частности, это касается точек останова на вызов функции API. Когда создаете такую точку останова, всегда при помощи условной конструкции указывайте, к какому процессу она относится. Для этого используйте функцию `PID`, которая возвращает текущий идентификатор процесса. Значение же идентификатора можно получить с помощью все той же команды `ADDR`. Пример создания условной точки останова на API-функцию `CreateWindowEx`:

```
: BPX CreateWindowEx if(PID==0x58)
```

Еще раз подчеркнем, что значение идентификатора для интересующего нас процесса можно определить при помощи команд `ADDR` или `PROC`. Для того чтобы установить такую точку останова, совсем не требуется делать интересующий нас процесс текущим.

Точки останова

При исследовании конкретного исполняемого кода одной задачи всегда является поиск нужного места в программе. При отсутствии текста на языке высокого уровня, а так почти всегда и бывает, если вы только не отлаживаете собственный программный проект, незаменимым средством являются *точки останова*.

Одноразовые точки останова

Одноразовые точки останова функционируют только один раз. Фактически такая точка останова — это строка в окне кода, на которую указывает курсор (подсвеченная строка). Перемещается курсор при помощи команды `U`. Команда `HERE` (или нажатие клавиши <F7>) выполняет исполняемый код, начиная с текущей команды и до отмеченной таким образом строки. Имейте в виду, что команда `HERE` набирается из окна кода, куда вы должны пред-

варительно перейти (клавиша <F6>). Можно также воспользоваться командой `G address`, и тогда код будет выполняться до адреса *address*.

Постоянные точки останова

Типичным примером *постоянной точки останова* является точка останова на конкретной команде (конкретный виртуальный адрес процесса). Для ее установки следует выйти в окно кода и использовать команду `BPX` без параметров. Вы можете двигаться по коду и устанавливать точки останова по нужным адресам. При этом строки, на которые устанавливаются точки прерывания, подсвечиваются. Точно такого же результата можно достигнуть, если воспользоваться клавишей <F9>. Убрать точку останова можно также повторным запуском команды `BPX` на уже поставленной точке останова или нажатием клавиши <F9>.

К данным точкам останова применима общая схема управления точками останова:

- `BL` — получить список точек останова с номерами;
- `BC` *n* — удалить точку останова с заданным номером;
- `BC *` — удалить все точки останова;
- `BE` *n* — редактирование точки останова с заданным номером.

Наконец, если вы знаете адрес, куда хотите поставить точку останова, то можно использовать этот адрес в команде `BPX`. Например, `BPX 0008:806CEFAB`. Разумеется, повторное использование команды `BPX` с тем же адресом удаляет точку останова. Не забывайте, что точка останова на адрес команды относится к конкретному адресному пространству, т. е. конкретному процессу.

Условные точки останова

В *условных точках останова* указываются те условия, при наступлении которых точка должна активизироваться. Невозможно поставить две точки останова на один адрес или на одну API-функцию, но вы можете при помощи условных конструкций учесть разные варианты вызова одной и той же точки останова. Типичные примеры использования условных точек останова представлены далее.

Пример 1

Точка останова на конкретный адрес срабатывает, только если содержимое регистра `EAX` принимает указанное значение.

```
: BPX 0008:806CEFAB if(EAX==406090)
```

Пример 2

Маленькое исследование с точкой останова на вызов функции `MessageBox` (рассмотрено приложение WinRar). После запуска приложения WinRar вызовем окно SoftICE и определим идентификатор приложения, используя команду `ADDR`. Идентификатор оказался равным `0x328`. Пишем следующую команду создания условной точки останова:

```
: BPX MessageBoxA if(PID==0x328)
```

Проверим командой `BL`, что точка останова задана, как и положено. Заметим, что мы указали суффикс `A` — это обязательно, так SoftICE различает функции API по их истинному имени.

Выйдем из отладчика, нажав клавишу <F5>, и выполним одну из команд программы, которая должна вызвать появление окна `MessageBox`. Тут же отобразится окно SoftICE. В командном окне появится сообщение о причине появления SoftICE. В данном случае мы видим:

```
Break due to BP 00: USER32!MessageBoxA IF(PID==0x328) (ET=2.65 seconds)
```

Обратим внимание теперь на окно кода. Там подсвечена первая строка входа в процедуру `MessageBox`:

```
USER32!MessageBoxA
001B:77D56471 CMP DWORD PTR [77D8C3D0],0
```

Теперь мы запросто можем исследовать стек вызова функции `MessageBoxA` и получить адрес возврата и значения параметров. Выполнив, например, команду `? *(ESP+4)`, получим значение дескриптора окна, которое и инициализировало вызов `MessageBox` (если вам это непонятно, то вернитесь к *разд. 3.2.1*). Значение `HWND` оказывается равным `100EC`. Просмотрев список окон приложения WinRar с помощью команды `HWND 328`, мы убеждаемся, что такое окно действительно есть и соответствует классу `WinRarWindow`. Кстати, здесь же в таблице мы видим и адрес функции данного окна, так что можем запросто углубиться в изучение работы этого окна.

Но вернемся снова к первой строке вызова функции `MessageBox` и найдем теперь адрес возврата. На адрес возврата, разумеется, указывает регистр `ESP`, и командой `? *(ESP)` мы получаем, что он равен `43C76D`.

Впрочем, адрес возврата можно получить и другим способом: нажать клавишу <F11> и после появления окна `MessageBox` и нажатия одной из кнопок этого окна мы опять оказываемся в SoftICE на строке, следующей за вызовом функции `MessageBox`.

Замечание

Вообще, поиск по вызову той или иной функции API — дело достаточно тонкое. Надо хорошо знать эти функции и понимать, что один и тот же результат дости-

12 Зак. 119

гается разными способами. Скажем, вам надо узнать, где создается окно. "Нужно искать CreateWindow", — скажете вы. А вот и нет.

Во-первых, в действительности функции CreateWindow нет. На самом деле, даже если вы в своей программе вызываете CreateWindow, то все равно используется CreateWindowEx.

Во-вторых, искать надо не CreateWindowEx, а CreateWindowExA и CreateWindowExW.

В-третьих, окно могло быть создано модальными диалоговыми функциями DialogBoxIndirect, DialogBoxParam, DialogBoxIndirectParam или немодальными функциями CreateDialogParam, CreateDialogIndirect, CreateDialogIndirectParam. Причем для всех функций надо учитывать суффиксы A и W.

Пример 3

Отслеживание содержимого регистров:

```
: BPX EIP IF(EAX==0x10)
```

Прерывание по данной точке останова сработает, когда значение регистра EAX будет равно 0x10, вне зависимости от того, в каком потоке будет происходить данное событие.

Прерывания на сообщения Windows

Как мы знаем, основное действо в приложениях GUI разворачивается в оконных функциях. Как реагирует функция на то или иное сообщение — важнейшая задача исследования. И вот тут незаменимую услугу может оказать точка прерывания на сообщение Windows. Вот пример установки такой точки:

```
: BMSG 100EC WM_CREATE
```

Первым параметром команды оказывается дескриптор окна, на функцию которого должно прийти сообщение. Отладчик по значению дескриптора окна определяет поток, который создал это окно, так что об этом можно не заботиться. По приходу нужного нам сообщения будет вызван отладчик SoftICE, а в окне кода будет показано начало функции окна. Интересно, что того же результата можно добиться и с помощью обычной команды BPX:

```
: BPX 43C76D IF((ESP->8)==WM_CREATE)
```

Первым параметром я указал адрес первой команды функции окна. А далее воспользовался тем фактом, что второй параметр функции находится на расстоянии 8 байтов от вершины стека.

Поиск процедуры окна

Итак, как же можно выйти на процедуру окна? Вот несколько простых советов.

- Просмотрите список окон приложения, который выводится командой `HWND n`, *n* — идентификатор приложения (идентификатор приложения, как вы уже знаете, можно получить с помощью команды `ADDR`). В списке окон имеются названия. Иногда по ним легко определить нужное окно и, соответственно, адрес процедуры.
- Список может оказаться небольшим, и вы легко можете протестировать все процедуры, поставив в начале процедуры (на одну из первых команд) точку останова. В случае если при активизации окна сработает точка останова, это и будет нужное нам окно.
- Проанализируйте работу окна на предмет того, какие функции API могут вызываться при работе с этим окном (так, в частности, мы поступили в примере 2 из предыдущего раздела). Поставьте точку останова на эту функцию и поработайте с этим окном. При прерывании определите, откуда вызывалась данная функция. Это и будет функция окна. Кроме этого, имейте в виду, что многие функции API первым параметром содержат дескриптор окна.

Если приложение содержит отладочную информацию

SoftICE — полноценный отладчик, т. е. он может загружать отладочную информацию и представлять ее вместе с исполняемым кодом. Таким образом, его можно использовать при отладке собственных приложений вместо стандартного отладчика, встроенного в интегрированную среду. Рассмотрим, например, как это делается при программировании на C++ в Visual Studio .NET.

При установке опции "добавить отладочную информацию" (для этого лучше всего выбрать конфигурацию проекта **DEBUG**) вместе с исполняемым модулем создается база отладочной информации. База представляет собой файл, по умолчанию имеющий то же имя, что и исполняемый модуль, и расширение pdb *(см. также разд. 1.6.1)*. Информации, хранящейся в файле, достаточно, чтобы представить структуру исходной программы вместе с именами глобальных и локальных переменных и сопоставить эту структуру машинному коду.

При загрузке исполняемого модуля при помощи загрузчика Loader32.exe загружается также отладочная информация и передается отладчику. По умолчанию, если для программы имеется отладочная информация, то SoftICE представляет в окне кода текст программы без ассемблерных команд. В дальнейшем при помощи команды `SRC` вы можете переключиться в смешанное представление программы (текст программы и машинный код)

или же к чисто машинному коду. В первом случае пошаговое выполнение программы означает ее пооператорное выполнение, в смешанном представлении шаг — это одна машинная команда. Соответственно точки останова можно устанавливать как на операторы языка высокого уровня, так и на машинные команды. Вот несколько строк, которые видны из окна кода SoftICE для случая смешанного представления:

```
00006               a=10;
001B:00411A2E       MOV DWORD PTR [EBP-a],0000000A
00007               b=11;
001B:00411A35       MOV DWORD PTR [EBP-b],0000000B
00008               c=10;
001B:00411A3C       MOV DWORD PTR [EBP-c],0000000C
00009               printf("%d\n",max(a,b,c));
001B:00411A43       MOV EAX,[EBP-c]
001B:00411A46       PUSH EAX
...
```

Разумеется, читатель понимает, что в записи типа `[EBP-a]` величина `a` — это адрес переменной `a` в стеке, точнее, смещение относительно адреса, где находится старое значение `EBP`, т. е. просто 4.

4.2. Краткий справочник по SoftICE

Справочник содержит бóльшую часть команд отладчика SoftICE, которых более чем достаточно для исследования исполняемого кода.

4.2.1. Горячие клавиши

Управление экраном

Управление экраном выполняется с помощью следующих комбинаций клавиш:

- <Ctrl>+<D> — вызов или закрытие главного окна SoftICE;
- <Ctrl>+<Alt>+<*стрелки*> — перемещение главного окна SoftICE на экране с шагом, равным размеру символа;
- <Ctrl>+<Alt>+<Home> — перемещение главного окна SoftICE в левый верхний угол экрана;
- <Ctrl>+<Alt>+<End> — перемещение главного окна SoftICE в левый нижний угол экрана;

- <Ctrl>+<Alt>+<PageUp> — перемещение главного окна SoftICE в правый верхний угол экрана;
- <Ctrl>+<Alt>+<PageDn> — перемещение главного окна SoftICE в правый нижний угол экрана;
- <Ctrl>+<L> — обновление главного окна SoftICE;
- <Ctrl>+<Alt>+<C> — размещение главного окна SoftICE в середине экрана.

Перемещение внутри главного окна

Перемещение внутри главного окна осуществляется с помощью следующих комбинаций клавиш:

- <Alt>+<C> — переход в окно кода из командного окна и обратно;
- <Alt>+<D> — переход в окно данных из командного окна и обратно;
- <Alt>+<L> — перемещение в окно локальных переменных из командного окна и обратно;
- <Alt>+<R> — перемещение в окно регистров из командного окна и обратно;
- <Alt>+<W> — перемещение в окно слежения из командного окна и обратно;
- <Alt>+<S> — перемещение в окно стека из командного окна и обратно.

Переход в любое (кроме окна сопроцессора) из окон главного окна отладчика можно осуществить также щелчком левой кнопки мыши в соответствующем окне.

Перемещение содержимого окон

Перемещение содержимого окон выполняется с помощью следующих клавиш:

- <↑> — перемещение на одну строку назад;
- <↓> — перемещение на одну строку вперед;
- <←> — перемещение на один символ влево;
- <→> — перемещение на один символ вправо;
- <PageUp> — перемещение на одну страницу назад;
- <PageDn> — перемещение на одну страницу вперед;
- <Home> — переход к первой строке кода;
- <End> — переход к последней строке кода.

Управление командным окном

Клавиша <Enter> выполняет завершение командной строки и выполнение набранной команды. SoftICE помнит 32 введенных команды. Перемещение по командам, находящимся в буфере, осуществляется клавишами <↑>, <↓>. При этом учитывается уже набранный в командной строке префикс. Например, если вы набрали букву B, то будут появляться только команды, начинающиеся на эту букву. Если вы находитесь в окне кода, то для просмотра буфера команд следует использовать комбинации клавиш <Shift>+<↑>, <Shift>+<↓>.

При редактировании командной строки используются следующие клавиши:

- <Home> — перевести курсор на начало командной строки;
- <End> — перевести курсор на конец командной строки;
- <Insert> — переключить режимы вставки/замены;
- <Delete> — удалить символ справа от курсора со сдвигом фрагмента строки влево;
- <Backspace> — удалить символ слева от курсора со сдвигом фрагмента строки влево;
- <←>, <→> — переместить курсор по строке.

Отладчик SoftICE имеет *буфер протокола окна команд*. Этот буфер содержит всю информацию, выводимую ранее в окне. Просмотреть содержимое буфера можно при помощи клавиш <PageDn> и <PageUp>.

Функциональные клавиши

Отладчик SoftICE позволяет использовать следующие функциональные клавиши:

- <F1> — выдать справочные сведения (равносильно команде H);
- <F2> — открыть/закрыть окно регистров;
- <F3> — переключиться между режимами исходного кода;
- <F4> — показать экран отлаживаемого приложения;
- <F5> — вернуться в отлаживаемую программу;
- <F6> — перевести курсор в окно кода или из него;
- <F7> — выполнить отлаживаемое приложение до команды, на которую указывает курсор;
- <F8> — выполнить текущую команду отлаживаемого приложения с заходом в функции;
- <F9> — установить точку останова на текущую команду;

- <F10> — выполнить текущую команду процессора с обходом функции;
- <F11> — перейти в вызывающую функцию программы;
- <F12> — выполнить функцию до выхода в вызывающую программу;
- <Shift>+<F3> — изменить формат вывода информации в окне данных;
- <Alt>+<F1> — открыть/закрыть окно регистров;
- <Alt>+<F2> — открыть/закрыть окно данных;
- <Alt>+<F3> — открыть/закрыть окно кода;
- <Alt>+<F4> — открыть/закрыть окно слежения;
- <Alt>+<F5> — очистить содержимое окна команд;
- <Alt>+<F11> — показать данные, которые расположены по адресу, размещенному в первом двойном слове окна данных;
- <Alt>+<F11> — показать данные, которые расположены по адресу, размещенному во втором двойном слове окна данных.

Замечание

Список команд отладчика, который можно получить при помощи нажатия клавиши <F1> или с помощью команды `H`, довольно обширен, но содержит не все команды. Полный список команд представлен в фирменном руководстве SoftICE Command Reference, которое можно найти на сайте **http://www.compuware.com** и других сайтах Интернета. Я в своей книге отталкиваюсь от списка, который выдает отладчик по команде `H`. Этих команд более чем достаточно, чтобы отлаживать и исследовать прикладные программы.

4.2.2. Команды SoftICE

Макрокоманды отладчика SoftICE

Команды, о которых мы будем говорить в данном разделе, могут объединяться в макрокоманды (макросы). Существуют два вида макрокоманд, которые могут выполняться в отладчике SoftICE.

Рассмотрим вначале макрокоманды *времени исполнения*. Эти команды существуют только в текущей сессии отладчика. После перезапуска они пропадают. Вот набор команд, с помощью которых можно управлять такими макрокомандами:

- `MACRO` *`имя_макрокоманды`* `=` `"`*`команда1;команда2;...`*`"` — создание или изменение макрокоманды. Например:

  ```
  : MACRO _ap "bc *;bpx MessageBox"
  ```

  создает макрокоманду с именем `_ap`;

- MACRO *имя_макрокоманды* * — удаление макроса с заданным именем. Например:

  ```
  : MACRO _ap *
  ```

  удаляет макрос _ap из списка макросов;
- MACRO * — удаление из списка всех макрокоманд;
- MACRO *имя_макрокоманды* — редактирование макрокоманды с данным именем;
- MACRO — вывод списка макрокоманд.

Можно определять макрокоманды с параметрами. Для этого используется символ %. После данного знака следует указать номер параметра. Номер должен лежать в диапазоне от 1 до 8. Например, команда

```
MACRO _bpx="bpx %1;bl"
```

создает макрос с именем _bpx с одним параметром. Данный макрос создает точку останова на указываемую в качестве параметра команду и выводит список точек останова. Для того чтобы вставить в определение макрокоманды знак " или %, следует использовать символ обратной косой черты \. Для вставки же косой черты предназначена последовательность \\.

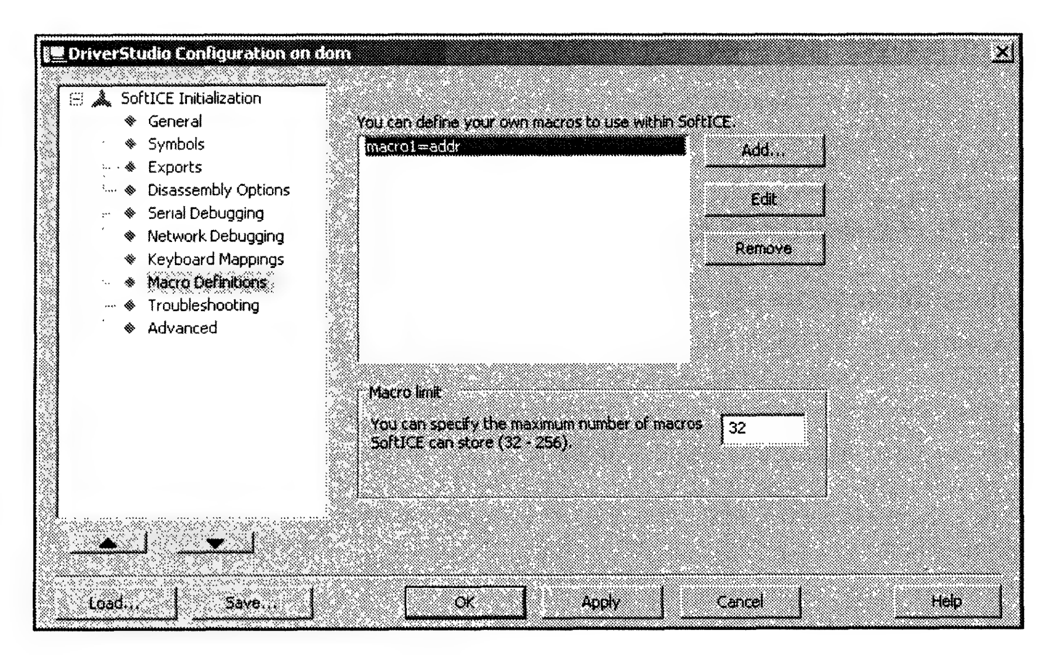


Рис. 4.4. Окно настройки создания постоянных макрокоманд

Для создания постоянных макрокоманд можно воспользоваться программой Loader32.exe. Для этого нам понадобится пункт меню **Edit | SoftICE Initialization Settings**. При выборе данного пункта появляется окно настроек SoftICE, в котором следует выбрать **Macro Definitions** (рис. 4.4). Дальнейшие действия довольно очевидны. Кнопки **Add** и **Edit** предназначены для добавления и редактирования макросов соответственно. Кнопка **Remove** — для удаления макроса.

Замечание

Запомните, что все изменения, которые вы делаете в окне настроек отладчика SoftICE, вступают в силу только после перезагрузки SoftICE.

Команды управления окнами SoftICE

Перечислю их:

- `Lines` *n* — команда задает количество строк в главном окне отладчика; значение *n* от 25 до 60;
- `Width` *m* — команда задает ширину главного окна отладчика в символах; значение *m* в промежутке от 80 до 160;
- `Set font` *n* — команда задает размер шрифта, используемого отладчиком; *n* может принимать значения 1, 2, 3;
- `Set origin` *x y* — с помощью данной команды можно задать положение левого верхнего угла главного окна на экране;
- `Set forcepalette [on | off]` — если значение параметра равно `on`, то блокируется изменение системной палитры цветов;
- `Color [c1 c2 c3 c4 c5]|[reset]` — задает цветовую гамму главного окна отладчика. Команда `Color reset` возвращает цветовую гамму к исходному состоянию, заданному по умолчанию. Однобайтовые параметры `c1`—`c5` задают цвет букв и фона соответствующего элемента главного окна отладчика. Первый полубайт задает цвет фона, второй — цвет букв:
  - `c1` — цвет основного фона и букв;
  - `c2` — цвет фона и букв для вывода изменившихся флагов (в окне регистров);
  - `c3` — цвет фона и букв для выделения текущей команды в окне кода;
  - `c4` — цвет фона и букв на панели подсказки;
  - `c5` — цвет фона и букв разделительных линий между окнами;
- команды открытия и закрытия окон:
  - `WC` — окно кода;

* WD — окно данных; может существовать одновременно несколько окон данных. Номер окна можно указать через точку, например, так: WD.3;
* WF — окно сопроцессора;
* WL — окно локальных переменных;
* WR — окно регистров;
* WW — окно слежения;
* WS — окно стека;
* WX — окно регистров MMX.

Каждая из перечисленных команд открывает или закрывает (если окно уже есть) соответствующее окно. При этом размеры главного окна не меняются, так что появление или удаление соответствующего окна идет за счет размеров уже имеющихся окон. Вы можете также задать размер (количество строк) окна, если укажете параметр в команде, например: WD 30 — команда задает количество строк в окне данных;

- EC — переход между окном команд и окном кода (эквивалентно использованию клавиши <F6>);
- CLS — по этой команде будет очищено командное окно (эквивалентно нажатию комбинации клавиш <Ctrl>+<F5>);
- RS — с помощью данной команды можно временно убрать с экрана окно SoftICE. При нажатии любой клавиши происходит восстановление окна SoftICE. Команда эквивалентна нажатию клавиши <F4>;
- ALTSCR — команда предназначена для перенаправление окна SoftICE на дополнительный монитор. Формат команды:

```
ALTSCR [mono | vga | off]
```

Назначение параметров:

* mono — монохромный монитор;
* vga — монитор, который поддерживает VGA-режимы;
* off — выключить альтернативный монитор (по умолчанию);

- FLASH — команда предназначена для восстановления экрана после команд T и P. Формат команды:

* включить режим восстановления

```
FLASH on
```

* выключить режим восстановления

```
FLACH off
```

При выполнении команды без параметров происходит вывод на экран в текущем режиме.

Получение и изменение информации в окнах

Список команд таков.

- R — команда получения и изменения информации, хранимой в регистрах. Формат команды:

  `R [-d|reg_name|reg_name [=] value]`

  Параметры:

  - `R -d` — просто выдает список регистров и их содержимое в окно команд;
  - `R reg_name` — переводит курсор в окне регистров в содержимое указанного в команде регистра. При этом вы можете редактировать содержимое, закрепляя исправление клавишей <Enter>. Разумеется, в окно регистров можно перейти другим способом, например, при помощи мыши, и точно также редактировать содержимое регистров;
  - `R reg_name = value` (знак `=` можно опустить) — заносит в указанный регистр значение *value*.

- U — вывод в командное окно дизассемблированного листинга. Формат команды:

  `U [address [L length]]`

  Параметры:

  - *address* — адрес, с какого предполагается вывод листинга. Можно указывать регистр, откуда этот адрес будет взят;
  - *length* — количество выводимых в листинге байтов (длина).

  При указании длины листинг выводится в командное окно. Если длину не указывать, но задать адрес, то это приводит просто к тому, что листинг в окне кода будет начинаться с указанного адреса. Команда без параметров приводит к прокрутке содержимого в окне кода, начиная с текущего адреса (на котором закончился предыдущий листинг). Если окно кода отсутствует, то весь вывод информации осуществляется в командное окно.

- D — команда вывода области памяти (дампа памяти). Формат команды:

  `D[size] [address [L length]]`

  Параметры:

  - *size* — размер, может принимать значения: B — побайтовый вывод, W — вывод словами, D — вывод двойными словами, S — вывод короткими вещественными числами (32 бита), L — вывод длинными вещественными числами (64 бита), T — вывод 10-байтовыми блоками;

- *address* — адрес, с какого предполагается вывод дампа. Можно указывать регистр, откуда этот адрес будет взят;
- *length* — количество выводимых в листинге байтов (длина). По умолчанию это значение равно 128.

Вывод осуществляется в окно данных. В случае если это окно отсутствует, дамп выводится в командное окно.

- E — команда редактирования памяти. Формат команды:

```
E[size] [address [data_list]]
```

Параметры:

- *size* — имеет тот же смысл и значение, что и для команды D;
- *address* — определяет адрес редактируемой области;
- *data_list* — при отсутствии данного параметра курсор переходит в окно данных, где вы можете непосредственно отредактировать ячейку памяти. В качестве этого параметра выступают данные, которые помещаются в ячейки памяти, начиная с указанного адреса. Формат данных должен соответствовать параметру *size*. Если значений несколько, то они должны отделяться друг от друга запятыми.

Пример использования команды:

```
EB EBX 33,34,35
```

По этой команде, начиная с адреса, который находится в регистре EBX, в три ячейки памяти будут помещены, соответственно, значения 33, 34, 35.

- PEEK — команда чтения непосредственно из физической памяти. Формат команды:

```
PEEK[size] address
```

Параметры:

- *size* — размер ячейки памяти, принимает значения: B — байт, W — слово, D — двойное слово;
- *address* — адрес, откуда производится чтение.

- POKE — команда записи непосредственно в физическую память. Формат команды:

```
POKE[size] address value
```

Параметры:

- *size* — имеет такой же смысл, как и для команды PEEK;
- *address* — физический адрес, куда осуществляется запись;
- *value* — записываемое в физическую память значение.

- `PAGEIN` — загрузка отсутствующей страницы в физическую память. Формат команды:

  ```
  PAGEIN address
  ```

  Параметром данной команды является виртуальный адрес страницы.
- `WATCH` — с помощью данной команды задается выражение, которое затем будет отслеживаться в окне слежения. Пример использования команды:

  ```
  WATCH ds:eax
  ```

  Таким образом, будут отслеживаться данные, адрес которых находится в регистре `EAX`.
- `FORMAT` — с помощью этой команды можно изменить формат вывода в окне данных. Команда не имеет параметров. Она просто циклически (по кругу) переводит содержимое окна из одного формата в другой.
- `DATA` — с помощью этой команды можно создавать дополнительные окна для просмотра данных. В качестве параметра команды можно использовать номер окна от 0 до 3.
- `A` — команда для ввода по указанному адресу ассемблерной команды. Формат команды:

  ```
  A [address]
  ```

  Единственным параметром команды является адрес, куда будет помещена вводимая вами ассемблерная команда. Если адрес не указывать, то берется текущий адрес из области кода. При выполнении команды в командном окне появляется подсказка (адрес), после чего вы можете ввести ассемблерную команду.
- `S` — команда поиска данных. Формат команды:

  ```
  S [-acu] [address L lenght data_list]
  ```

  Параметры:

  - `c` — поиск без учета регистра;
  - `u` — поиск в формате Unicode;
  - `a` — поиск в формате ASCII;
  - *`address`* — начальный адрес поиска;
  - *`length`* — размер охватываемой поиском области памяти;
  - *`data_list`* — перечень данных для поиска, отделенных друг от друга запятыми или пробелами.

  Команда предназначена для поиска нужных данных. В случае их обнаружения они будут отображены в окне данных, а в командном окне появятся соответствующие сообщения с указанием адреса расположения. Для

продолжения поиска следует ввести эту команду без параметров. Пример поиска байта `20h` в области длиной `2000h`, начинающейся с адреса, который хранится в регистре `EAX`:

```
S ds:eax L 2000 20
```

- `F` — команда заполнения области памяти. Формат команды:

  `F` *`address`* `L` *`length data_list`*

  Параметры:

  - *`address`* — начальный адрес;
  - *`length`* — длина заполняемой области;
  - *`data_list`* — данные, которые будут помещены, начиная с указанного адреса, разделенные запятыми либо пробелами.

  Команда помещает данные, указанные в *`data_list`*, начиная с заданного адреса. Если *`length`* больше длины данных, они будут циклически повторены до достижения размера `length`. Например, область, которая начинается по адресу `DS:EAX` и имеет длину `100H`, будет заполнена символами `W`:

  ```
  F ds:eax L 100 "W"
  ```

- `M` — команда перемещения данных. Формат команды:

  `M` *`address1`* `L` *`length address2`*

  Параметры:

  - *`address1`* — адрес, откуда будут переноситься данные;
  - *`length`* — длина переносимых данных;
  - *`address2`* — адрес, куда будут переноситься данные.

  Пример команды:

  ```
  M ds: eax L 1000 ds:ebx
  ```

  По данной команде `1000h` байтов будет перенесено из адреса, на который указывает `EAX`, в область с адресом, который хранится в регистре `EBX`.

- `C` — команда сравнения двух блоков данных. Формат команды:

  `C` *`address1`* `L` *`length address2`*

  Параметры:

  - *`address1`* — адрес первого сравниваемого блока;
  - *`length`* — длина сравниваемых данных;
  - *`address2`* — адрес второго блока данных.

С помощью этой команды можно сравнить два блока данных. Если будет обнаружено неравенство, то в командном окне будут выведены полные адреса этих байтов и их значения. Например:

```
C ds:100 L 10 ds:200
```

Будет произведено сравнение `10h` байтов.

- `HS` — данная команда может быть использована для поиска в командном буфере. Формат команды:

  `HS [+|-]` *`string`*

  Знаки `+` и `-` определяют, соответственно, нисходящий (сверху вниз) и восходящий (снизу вверх) поиск. Далее указывается строка для поиска. Для продолжения поиска следует запустить команду без параметров.

- Команда `.` — точка. Если окно кода видимо, то данная команда делает инструкцию по адресу `CS:EIP` видимой и подсвечивает ее.

Команды управления точками останова

Точки останова, или точки прерывания, являются важнейшим механизмом отладки приложений. SoftICE присваивает каждой точке прерывания номер от 0 до 255. Таким образом, всего одновременно могут существовать 256 точек прерывания. С помощью этого номера можно управлять точками прерывания: удалять и включать/выключать. Количество точек останова на обращение к памяти и портам ввода/вывода в сумме не должно превышать 4.

Типы точек прерывания

Перечислим основные точки прерывания, которые поддерживает отладчик SoftICE.

- Прерывание по исполняемым командам. В этом прерывании можно задавать имя, при появлении которого в исполняемом коде должна произойти остановка выполнения кода. В частности, вы можете установить точку останова на вызов какой-либо функции API.
- Прерывание на обращение к памяти. По данному прерыванию отладчик следит за обращением по определенному адресу памяти.
- Точки останова прерываний. Отладчик будет отслеживать прерывания, происходящие в операционной системе, путем модификации таблицы IDT *(см. разд. 1.2.1)*.
- Прерывания на команды ввода/вывода. Отладчик отслеживает инструкции `IN`/`OUT`.
- Прерывания на сообщения Windows. При этом надо знать дескриптор окна, куда должно прийти данное сообщение.

Возможности точек прерывания

При работе с точками прерывания можно использовать условные конструкции. Точка останова сработает только тогда, когда указанное условие будет выполнено. В частности, с помощью условия можно определить, для какого процесса будет срабатывать данная точка останова. Типичный пример такого условия `if(pid==0x058)` — условие того, что идентификатор процесса должен быть равен значению `0x058`. Этим условием нам придется пользоваться постоянно, поскольку мы будем отлаживать конкретные запущенные приложения.

При помощи оператора `do` можно задать команды, которые будут выполняться, если сработает точка останова. В общем случае формат команды выполнения действий имеет вид:

```
do "команда1;команда2;..."
```

В качестве команд могут выступать обычные команды отладчика или макрокоманды.

Ниже при описании команд, используемых при работе с точками останова, будут применяться следующие обозначения.

- *size* — определяет размер ячейки, на которую будет устанавливаться точка прерывания. Может принимать значения: `B` — байт, `W` — слово, `D` — двойное слово;
- параметр `[R|W|RW|X]` определяет тип доступа к ячейке памяти и порту ввода/вывода, который будет отслеживаться. `R` — чтение из ячейки (порта), `W` — запись в ячейку (порт), `RW` — чтение и запись в ячейку (порт), `X` — выполнение команды, занимающей данную ячейку памяти;
- *Reg_deb* — здесь можно указать, какой регистр отладки следует использовать (`D0`—`D3`). Как правило, это не делают, т. к. отладчик выбирает нужный регистр;
- `[IF` *cond*`]` — здесь нужно указать условие, которое должно выполниться, чтобы было возможно прерывание по данной точке останова;
- `[DO` *comm*`]` — можно указать команду или группу команд, которые будут выполняться при прерывании в данной точке.

Команды установки точек прерывания

Перечислю их.

- `BPM` — с помощью данной команды можно установить точку прерывания на ячейку памяти. Формат команды:

  ```
  BPM[size] addr [R|W|RW|X] [reg_deb] [IF cond] [DO comm]
  ```

  Параметр *addr* определяет адрес ячейки. Адрес можно указать явно или посредством регистров, например, `ds:eax`.

- BPIO — данная точка останова устанавливается на ввод/вывод в указанный порт. Формат команды:

  ```
  BPIO [R|W|RW] [deb_reg] [IF cond] [DO comm]
  ```

  Отладчик будет отслеживать все команды ввода/вывода в указанный порт.

- BPINT — данная команда используется для установки точки останова на прерывание. Точка останова срабатывает только в том случае, если прерывание срабатывает через IDT (таблицу дескрипторов прерываний). Формат команды:

  ```
  BPINT int_number [IF cond] [DO comm]
  ```

  Здесь `int_number` — номер отслеживаемого прерывания. При срабатывании точки останова первой инструкцией будет первая команда обработчика прерываний.

- BPX — эта команда устанавливает точку останова на выполнение, например, на выполнение какой-либо функции API. Формат команды:

  ```
  BPX exp [IF cond][DO comm]
  ```

  Здесь *exp* — некоторое имя. Пример:

  ```
  BPX MessageBoxW
  ```

  Команда BPX, не содержащая параметров, устанавливает точку останова на текущую команду, но для этого следует перейти в окно кода отладчика.

- BMSG — команда предназначена для установки точки прерывания на сообщения, приходящие для определенного окна в конкретном диапазоне. Формат команды:

  ```
  BMSG hWnd [L] [beg_mes [end_mes]][IF con] [DO comm]
  ```

  Параметры:

  - *hWnd* — дескриптор окна;
  - L — при установке этого параметра сообщение будет отображено в буфере (окне) команд, а сам отладчик не будет активизирован;
  - beg_mes — первое сообщение диапазона сообщений. Параметр может быть задан как числовым, так и символьным обозначением сообщения;
  - *end_mes* — последнее сообщение диапазона (если речь идет именно о диапазоне, а не об одном сообщении). Если данный параметр отсутствует, то отлавливается лишь сообщение, заданное параметром *beg_mes*.

Если сообщения в команде не указывать, то точка останова накладывается на все сообщения данного окна. Пример использования команды:

```
BMSG 01001F WM_PAINT
```

Перехват сообщения `WM_PAINT` для окна с дескриптором `01001F`.

- `BSTAT` — данная команда служит для выдачи статистической информации по заданной точке останова. В качестве параметра команды следует указать номер точки прерывания. В частности, будет выдана величина `Popups` — количество раз, когда данная точка прерывания вызывала окно SoftICE, `Breaks` — количество срабатываний точки останова и т. д.

Команды манипулирования точками прерывания

Список таков.

- `BPE` — команда редактирования точки останова. Параметром данной команды служит номер точки останова.
- `BPT` — данная команда вызывает в командную строку шаблон создания точки останова с заданным номером. Отличие от предыдущей команды заключается в том, что с помощью данной команды создается еще одна точка останова, а не редактируется уже существующая.
- `BL` — данная команда выдает список точек останова — номер и шаблон создания.
- `BC` — команда удаления точки останова. Параметром данной команды может служить номер точки останова или список номеров (через запятую или пробел), подлежащих удалению. Если в качестве параметра использовать символ `*`, то будут удалены все точки останова.
- `BD` — команда приостанавливает работу точек останова. В качестве параметра данной команды может быть список точек останова (номера через запятую или пробел) или символ `*`.
- `BE` — команда возобновляет работу точек останова. В качестве параметра данной команды может быть список точек останова (номера через запятую или пробел) или символ `*`.
- `BH` — выдает список точек останова, которые использовались в данной и предыдущей сессии работы отладчика. Продвигаясь по появившемуся списку, можно с помощью клавиши <Insert> выбрать точку останова, которую вы хотите использовать сейчас. Клавиша <Enter> служит для установки всех выбранных точек останова. Отладчик помнит последние 32 точки останова.

Команды трассировки

Перечислю их.

- X — выход из окна SoftICE и возвращение управления программе, прерванной вызовом SoftICE. Равносильно нажатию клавиши <F5> или комбинации клавиш <Ctrl>+<D>.
- G — команда сообщает отладчику, что необходимо выполнить отлаживаемое приложение. Формат команды:

  ```
  G [=address1] [address2]
  ```

  Параметры:

  - *address1* — адрес, с которого должно начаться выполнение. Если данный адрес не указан, то выполнение начнется с текущего адреса (CS:EIP);
  - *address2* — конечный адрес выполнения. Если данный адрес не указан, то выполнение будет происходить до тех пор, пока не встретится точка останова или не будет выполнен вызов окна SoftICE.

  Команда G без параметров равносильна команде X. Команда G @SS:EBP равносильна нажатию клавиши <F11> — перейти в вызывающую функцию.
- T — команда пошагового выполнения отлаживаемого кода. Формат команды:

  ```
  T [=address] [count]
  ```

  Параметры:

  - *address* — начальный адрес трассировки. Если данный адрес не указан, то выполнение начинается с текущей команды;
  - *count* — указывает, сколько инструкций следует выполнить. Если параметр отсутствует, то выполняется одна команда.

  Команда без параметров равносильна нажатию клавиши <F8>. Пример команды:

  ```
  T: T CS:EIP-20 10
  ```

  Будет выполнено 10 инструкций, начиная с адреса CS:EIP-20.
- P — выполнение инструкции с обходом вызова процедур, прерываний, а также строковых команд и циклов. Без параметров команда равносильна нажатию клавиши <F10>. Если присутствует опция RET (P RET), то SoftICE будет выполнять программу до обнаружения инструкций RETN/RETF, причем остановка будет там, куда произойдет переход с помощью этих команд. Таким образом, с параметром команда равносильна нажатию клавиши <F12>.

- HERE — данная команда равносильна нажатию клавиши <F7> — выполнить программу с адреса CS:EIP и до текущего положения курсора в окне кода.
- EXIT — считается устаревшей командой. Фактически равносильна команде X. Следует избегать использования этой команды.
- GENINT — передача управления прерыванию. Формат команды:

  ```
  GENINT [nmi | int1 | int3 | number]
  ```

  Параметры:

  - *nmi* — вызов немаскируемого прерывания;
  - *int1* — вызов прерывания номер 1;
  - *int3* — вызов прерывания номер 3;
  - *number* — вызов прерывания с номером от 0 до 5F.

  Использовать данную команду следует очень осторожно. Вы должны быть уверены, что обработчик прерывания существует, в противном случае команда вызовет зависание системы.
- HBOOT — команда осуществляет сброс (перезагрузку) компьютерной системы.
- I1HERE — имеет две формы:
  - I1HERE on — включение режима, окно отладчика SoftICE будет вызываться каждый раз, когда возникнет прерывание с номером 1;
  - I1HERE off — выключение режима.
- I3HERE — имеет две формы:
  - I3HERE on — включение режима, окно SoftICE будет вызываться каждый раз, когда возникнет прерывание с номером 3;
  - I3HERE off — выключение режима.
- ZAP — данная команда заменяет вызовы прерываний с номерами 1 и 3 на инструкции NOP.

Основные информационные команды

Их список таков.

- GDT — команда для отображения таблицы GDT. Формат команды:

  ```
  GDT [selector | address]
  ```

  Параметры:

  - *selector* — селектор в таблице GDT;
  - *address* — адрес сегмента.

Если не указывать параметры, то будет отображено содержимое всей таблицы GDT.

- `LDT` — команда для отображения таблицы LDT. Формат команды:

  `LDT [`*`selector`*` | `*`table_selector`*`]`

  Параметры:

  • *`selector`* — селектор в LDT;

  • *`table_selector`* — селектор LDT в GDT.

  Команда без параметров выдает всю таблицу LDT.

- `IDT` — команда для отображения содержимого таблицы прерываний. Формат команды:

  `IDT [`*`number`*` | `*`address`*`]`

  Параметры:

  • *`number`* — номер прерывания, информацию о котором из таблицы IDT следует отобразить;

  • *`address`* — адрес обработчика прерываний (селектор:смещение), информацию о котором из таблицы IDT следует отобразить.

  Без параметров команда выводит текущее содержимое всей таблицы IDT.

- `TSS` — по данной команде в командном окне будет выведено содержание сегмента TSS. Параметром команды является селектор в GDT, указывающий на TSS. Если команду запустить без параметра, то будет показано содержание текущего TSS, селектор которого находится в регистре задач `TR`.

- `CPU` — на данную команду выдается полный список регистров процессора и их содержимое.

- `PCI` — команда выводит в командном окне информацию обо всех PCI-устройствах, имеющихся в системе.

- `MOD` — по данной команде в окно команд выдается список всех подключенных модулей Windows. В командной строке можно указать первые буквы имени модуля, тогда будет выдан список модулей, имена которых начинаются с указанного префикса.

- `HEAP32` — выдает список системных и созданных приложениями куч (heaps) памяти. Формат команды:

  `HEAP32 [`*`hheap`*` | `*`name`*`]`

  Параметры:

  • *`hheap`* — дескриптор кучи, возвращаемый функцией `CreateHeap`;

  • *`name`* — имя задачи.

Команда выдает базовый адрес кучи, максимальный размер, количество килобайт используемой памяти, количество сегментов в куче, тип кучи, владельца кучи. В отсутствие параметров команда выдает список всех куч.

- `TASK` — по данной команде в командном окне будет отображен весь список задач и дополнительная информация о них. Возле активной задачи будет символ `*`. Команда может быть полезна в случае сбоя в системе для определения задачи, вызвавшей его.
- `NTCALL` — команда выдает список системных сервисов, функционирующих на уровне ядра (кольцо 0).
- `WMSG` — выдает в командное окно список сообщений Windows и их номера. Формат команды:

  `WMSG [partial_name] [number]`

  Параметры:

  - *partial_name* — полное или частичное название сообщения;
  - *number* — номер сообщения Windows.

  Команда без параметров выводит список всех известных отладчику сообщений Windows. При наличии параметра *partial_name* выдаются все сообщения, соответствующие данному фрагменту имени сообщения. Если указан номер сообщения, то будет выдан номер и имя сообщения.
- `PAGE` — по этой команде будет выдана информация о страницах, начинающихся с данного виртуального адреса (виртуальный и физический адреса, атрибут, тип, виртуальный размер). Формат команды:

  `PAGE [address] [L length]`

  Параметры:

  - *address* — виртуальный адрес страниц;
  - *length* — количество выдаваемых страниц.

  Команда без параметров выдает список всех страниц.
- `PHYS` — данная команда вызывает отображение списка всех виртуальных адресов, соответствующих указанному физическому адресу. Команда используется только с параметром — физическим адресом.
- `STACK` — выдает информацию о структуре стека. Формат команды:

  `STACK [thread | frame]`

  Параметры:

  - *thread* — дескриптор или идентификатор потока;
  - *frame* — адрес стекового фрейма.

Команда без параметров выдает информацию о текущем стеке на основе адреса SS:EBP.

- XFRAME — выдает записанную в стек информацию об исключении *(см. разд. 3.2.5)*. Параметром команды служит идентификатор потока или указатель на фрейм стека. Если параметр отсутствует, то отладчик использует текущий поток.
- HWND — команда выдает информацию о созданных в системе окнах. Формат команды:

  ```
  HWND [-x] [-c] [hwnd | desktop | process | thread | module | class]
  ```

  Параметры:

  - -x — вывод расширенной информации;
  - -c — заставляет отладчик выдавать иерархию окон;
  - *hwnd* — дескриптор окна или указатель на структуру окна;
  - *desktop* — дескриптор рабочего стола;
  - *process* — идентификатор процесса;
  - *thread* — идентификатор потока;
  - *module* — имя модуля;
  - *class* — имя зарегистрированного класса окон.

  Команда без параметров выдает информацию обо всех созданных на данный момент в системе окнах.
- CLASS — выдает информацию о классах окон. Формат команды:

  ```
  CLASS [-x] [process] [thread] [module] [class]
  ```

  Параметры:

  - -x — выдавать расширенную информацию о классах;
  - *process* — идентификатор процесса;
  - *thread* — идентификатор потока;
  - *module* — идентификатор или имя модуля;
  - *class* — имя зарегистрированного класса.

  Команда без параметров выдает список всех зарегистрированных классов текущего процесса.
- THREAD — команда используется для получения информации о потоках. Формат команды:

  ```
  THREAD [-r | -x | -u] [thread] [process]
  ```

Параметры команды:

- `-r` — выдавать команду о регистрах потока;
- `-x` — выдавать расширенную информацию о потоках;
- `-u` — выдавать информацию о компонентах потока пользовательского уровня;
- *`thread`* — идентификатор потока;
- *`process`* — идентификатор процесса.

- `ADDR` — используется для выдачи информации о существующих адресных контекстах (процессов) и установлении текущего контекста. Для установления текущего контекста параметром команды следует указать идентификатор, имя процесса или адрес. Можно также указать адрес информационного блока процесса (KPEB, Kernel Process Environment Block — блок описания процесса уровня ядра). Всю эту информацию можно получить, если использовать команду `ADDR` без параметров.
- `MAP32` — выдает список загруженных 32-битных модулей и дополнительную информацию о них. Формат команды:

```
MAP32 [-u | -s] [name | handle | address]
```

Параметры:

- `-u` — показать только модули, загруженные в часть оперативной памяти, которую можно использовать для хранения программ пользователя;
- `-s` — показать только модули, загруженные в часть оперативной памяти, отведенную под операционную систему, и требуемые ей средства;
- *`name`* — имя модуля;
- *`handle`* — адрес модуля;
- *`address`* — адрес, принадлежащий модулю.

Команда без параметров выдаст список всех загруженных 32-битных модулей и дополнительную информацию о них.

- `PROC` — команда предназначена для получения информации о процессе. Формат команды:

```
PROC [-xom] [name]
```

Параметры:

- `-x` — показать расширенную информацию о каждой ветви;
- `-o` — показать расширенную информацию о каждом объекте;
- `-m` — показать информацию об использовании объектом памяти;

- *name* — имя задачи, имя процесса, дескриптор процесса, идентификатор процесса, имя потока, идентификатор потока, дескриптор потока.

 Если не указано имя объекта, то выводится информация обо всех процессах системы.

- QUERY — команда предназначена для вывода карты виртуальной памяти процессов. Формат команды:

 ```
 QUERY [-x] [address] [name]
 ```

 Параметры команды:

 - -x — показать имена процессов (и дополнительную информацию о них), которые занимают указанный виртуальный адрес;
 - *address* — виртуальный адрес;
 - *name* — имя процесса.

 Без параметров команда выдает карту виртуальной памяти текущего процесса.

- WHAT — данная команда пытается интерпретировать указанный в ней параметр. Например, если параметр — это идентификатор процесса, то команда сообщает об этом, т. е. вы тем самым можете проверить подлинность идентификатора или дескриптора.
- OBJTAB — команда для получения информации о таблице объектов USER.
- FOBJ — выдает информацию о файловых объектах, существующих в настоящее время. Такие объекты создаются для каждого открытого файла.
- IRP — команда выдает информацию об IRP (I/O request packet, пакет запроса ввода/вывода).
- FIBER — выдает структуру данных для волокон. Эта структура данных, в частности, возвращается функцией CreateFiber.

Другие команды

Список остальных команд таков.

- PAUSE — устанавливает два режима просмотра информации в командном окне:
 - PAUSE on (по умолчанию) — информация выдается порциями, следующая порция появляется по нажатию любой клавиши;
 - PAUSE off — информация выдается непрерывно.
- ? — команда вычисления выражения. Например, ? 34+90*2. Отладчик при этом выводит результат одновременно в шестнадцатеричной и десятичной системах, а также в ASCII-формате.

- OPINFO — получить информацию об инструкции процессора. Например, по команде OPINFO add на экран будет выдана основная информация об инструкции процессора ADD:

```
ADD
    Integer addition: DEST <- DEST + SRC
    EFLAGS | OF DF IF SF ZF AF PF CF TF NT RF |
           | M           M  M  M  M  M         |
```

- ALTKEY — данная команда производит изменение комбинации клавиш, используемых для активизации SoftICE. По умолчанию используется комбинация клавиш <Ctrl>+<D>. Если эту команду запустить без параметров, то SoftICE отобразит в командном окне текущую комбинацию. Примеры использования команды: ALTKEY Alt P, ALTKEY Ctrl Z — теперь окно SoftICE будет вызываться нажатием клавиш <Ctrl>+<Z>.

Операторы

В среде отладчика SoftICE в командах и определении условных точек останова можно использовать выражения. Для построения выражений можно применять операторы. Рассмотрим перечень используемых в отладчике операторов.

Операторы адресации

К этой категории относятся следующие операторы:

- . — точка. Если окно кода видимо, то данный оператор делает инструкцию по адресу CS:EIP видимой и подсвечивает ее. Точку можно использовать в выражениях;
- * — данный оператор используется для задания адреса, на который указывает данное выражение. Например, *(EAX) означает содержимое памяти, на которое указывает регистр EAX.
- -> — с помощью данного оператора, также как и с помощью оператора *, можно получить содержимое по адресу, на который указывает данное выражение. Например, если вам известен адрес процедуры окна, который, в частности, можно получить при помощи команды HWND, то допустимо использовать следующую точку останова на сообщение WM_PAINT:

```
BPX 6BDFE003 IF (ESP->8)==WM_PAINT
```

- @ — фактически эквивалентен оператору *.

Математические операторы

Список математических операторов таков:

- + — унарный и бинарный операторы; например, +100 или EBX+ESI;
- - — унарный и бинарный операторы; например, -100 или EAX-8;
- * — бинарный оператор умножения; например, EBX*4;
- / — бинарный оператор деления; например, (EAX+EBX)/2;
- % — бинарный оператор деления по модулю; например, EBX%3;
- << — оператор логического сдвига влево;
- >> — оператор логического сдвига вправо.

Побитовые операторы

- & — побитовый оператор "И";
- | — побитовый оператор "ИЛИ";
- ^ — побитовый оператор "исключающее ИЛИ";
- ~ — побитовый оператор инверсии или NOT.

Логические операторы

К данной категории относятся операторы:

- ! — логическое отрицание (NOT); например, !EBX;
- && — логическое "И"; например, EAX&&EBX;
- || — логическое "ИЛИ"; например, EAX||FF;
- == — условие равенства;
- != — условие неравенства;
- < — меньше;
- > — больше;
- <= — меньше или равно;
- >= — больше или равно.

Встроенные функции SoftICE

Дизассемблер имеет ряд собственных встроенных функций. Вот перечень основных функций:

- Byte — возвращает младший байт выражения;
- Word — возвращает младшее слово выражения;

- Dword — возвращает двойное слово (расширяет байт или слово);
- HiByte — возвращает старший байт (слова или двойного слова);
- HiWord — возвращает старшее слово;
- Sword — преобразует байт в слово со знаком;
- Long — преобразует байт или слово в длинное целое;
- WSTR — показывает строку в формате Unicode;
- Flat — преобразовывает адрес с селектором (логический адрес) в линейный адрес плоской модели памяти.

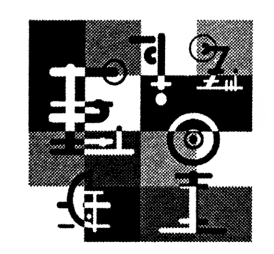
Текущее содержимое регистров может быть найдено при помощи функций, с названием, соответствующим названию регистра, например, EAX, EBX, EDX и т. д.:

- CFL — возвращает флаг переноса;
- PFL — возвращает флаг четности;
- AFL — возвращает флаг вспомогательного перехода;
- ZFL — возвращает флаг нуля;
- SFL — возвращает знаковый флаг;
- OFL — возвращает флаг переполнения;
- RFL — возвращает флаг возобновления;
- TFL — возвращает флаг трассировки;
- DFL — возвращает флаг направления;
- IFL — возвращает флаг разрешения прерывания;
- NTFL — возвращает флаг вложенной задачи;
- IOPL — возвращает уровень привилегий ввода/вывода;
- VMFL — возвращает флаг режима виртуального процессора;
- IRQL — возвращает текущий IRQ;
- DataAddr — возвращает начальный адрес блока данных, отображаемых в окне данных;
- CodeAddr — возвращает адрес первой инструкции, отображаемой в окне кода;
- Eaddr — возвращает эффективный адрес текущей инструкции, если такая есть;
- Evalue — возвращает значение по текущему эффективному адресу;
- Process — возвращает блок среды активного процесса;
- Thread — возвращает блок среды активного потока;

- PID — идентификатор активного процесса;
- TID — идентификатор активного потока;
- BPCount — возвращает количество попаданий в точку прерывания, при которых значение условного выражения равно true;
- BPTotal — возвращает общее количество попаданий в точку прерывания;
- BPMiss — возвращает количество попаданий в точку прерывания, при которых условие прерывания не было выполнено и окно SoftICE не активизировалось;
- BPLog — сохраняет в буфере информацию о попадании в точку прерывания;
- BPIndex — возвращает номер текущей точки останова в общем списке точек.

Если вы предваряете символом подчеркивания имя функции в каком-либо выражении, то дизассемблер вычисляет значение функции на данный момент и использует это значение в дальнейших вычислениях. Например: _PID, _TID, _EAX и т. д.

Глава 5



Дизассемблер IDA Pro

Дизассемблер IDA Pro — действительно выдающийся инструмент исследования исполняемого кода.

Три кита, на которых держится исследование кода в IDA Pro, — это:

- мощное средство анализа исполняемого кода, встроенное в дизассемблер. IDA Pro никогда не делает слишком "самоуверенных" предположений. Привилегия на эвристический анализ предоставляется человеку;
- человеку предоставляется возможность участвовать в этом анализе, уточнять параметры тех или иных объектов программы, делать исправления. Другими словами, пользователь данного инструмента становится активным участником процесса дизассемблирования;
- встроенный язык программирования, весьма близкий по своей структуре к классическому языку C, позволяет значительно наращивать функциональность данного продукта.

На протяжении всей книги мы пользовались этим замечательным инструментом, так что данная глава будет служить двум основным целям:

- более подробно описать основные возможности дизассемблера IDA Pro.
- представить справочный материал по данной программе, так чтобы читатель смог, по крайней мере, в начале обучения исследовать исполняемый код, заглядывая время от времени в мою книгу.

К сожалению, информация по данному отладчику крайне скудна, кроме довольно бедного справочного файла (idahelp.hlp) фактически ничего нет. И я надеюсь данной главой оказать помощь многочисленным желающим овладеть этим инструментом.

5.1. Введение в IDA Pro

IDA — это Interactive DisAssembler, а совсем не имя женщины, хотя в окне **About** и помещено изображение прекрасной женской головки. Но инструмент действительно изящен, так что его название (точнее ассоциации, которые оно вызывает) вполне соответствует его сути.

5.1.1. Начало работы

Прежде всего, замечу, что в состав пакета IDA Pro входит консольный (idaw.exe) и графический вариант (idag.exe) программы. В дальнейшем все рассмотрение интерфейса будет касаться именно графического варианта программы.

Общие сведения о виртуальной памяти

Если вы загрузите в IDA Pro некоторый исполняемый модуль, то в каталоге, откуда произошла загрузка, обнаружите два файла с расширениями id0 и id1. Это вспомогательные файлы виртуальной памяти, которые используются IDA Pro для хранения используемых им данных. При выгрузке загруженного модуля (**File | Close**) оба файла исчезают. В файл с расширением id1 и именем исследуемого модуля загружается образ этого модуля. Этот образ вполне идентичен образу, загруженному в 32-битную плоскую память операционной системой Windows. Таким образом достигается полная идентичность исследуемого модуля с модулем, исполняемым операционной системой, что, несомненно, сближает IDA Pro с отладчиками. Для каждого адреса в файле хранится 32-битная характеристика: 8-битная ячейка, соответствующая данному адресу, и 24-битный атрибут, определяющий различные свойства данной ячейки (а именно, относится ли данная ячейка к инструкции или к данным (и какой тип данных), а также есть ли другие объекты в строке: комментарии, перекрестные ссылки, метки).

Механизмы работы с виртуальной памятью IDA Pro вполне идентичны аналогичным механизмам, которые используются операционной системой Windows. При обращении к конкретной ячейке загружается в оперативную память (в буфер) вся страница, где эта ячейка расположена. Если же изменить ячейку памяти, то происходит перезапись всей страницы виртуальной памяти. Часть страниц IDA Pro держит в оперативной памяти. Модифицированные страницы периодически сбрасываются дизассемблером на диск. В случае, когда требуется загрузить страницу, а буфер страниц полон, IDA Pro ищет среди загруженных страниц модифицированную раньше всех, сбрасывает ее на диск и загружает на ее место требуемую страницу.

Кроме хранения образа загружаемого модуля IDA Pro требуется память для хранения вспомогательной информации: имен меток, имен функций и комментариев. Для этого используется файл с расширением id0. Эту память в документации называют *memory for b-tree*[1].

Интерфейс программы

Общие сведения

На рис. 5.1 представлено главное окно дизассемблера IDA Pro с загруженной туда исполняемой программой. Фоновый анализ уже закончился, о чем говорит сообщение в левом нижнем углу: "The initial autoanalysis is finished".

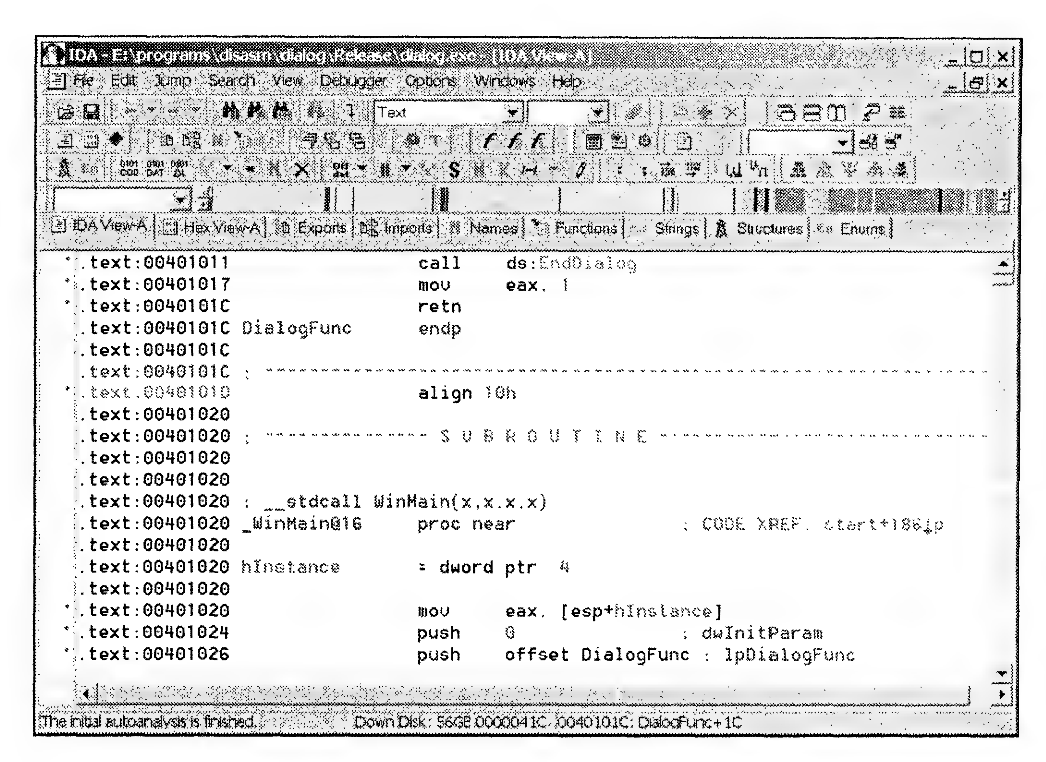


Рис. 5.1. Интерфейс главного окна программы IDA Pro

В окне IDA Pro мы видим большое количество вкладок. По умолчанию их девять. На самом деле количество вкладок может быть и больше. Их можно добавлять при помощи пунктов меню **Views | Open subviews**. Два окна —

[1] Balanced tree — дерево, у которого разность расстояний от корня до любых двух листьев не превышает 1.

IDA View (Окно дизассемблера) и **Hex View** (Окно дампа) — могут дублироваться: таким образом в разных окнах можно просматривать разные участки кода и данных. Эти окна имеют суффиксы, чтобы отличать их друг от друга: **A**, **B**, **C** и т. д.

Конечно, главным окном программы является **IDA View**. Именно в нем представлен основной результат анализа исполняемого кода, и где вы также можете поучаствовать в продолжение этого анализа.

Когда вы работаете с отладчиком IDA Pro, не забывайте о трех способах управления им: пункты меню, кнопки панели инструментов и горячие клавиши. Последние реализуют далеко не все возможности IDA Pro, но охватывают наиболее часто используемые операции. Например, если какой-то блок данных вызывает у вас сомнение, вы всегда можете преобразовать его в код (дизассемблировать), нажав клавишу <C> (от англ. *Code* — код). И наоборот, если последовательность ассемблерных команд кажется вам слишком бессмысленной, вы можете преобразовать ее в данные при помощи клавиши <D> (от англ. *Data* — данные).

IDA Pro использует следующие файлы конфигурации: ida.cfg — общий файл, idatui.cfg — файл конфигурации для консольного варианта программы, idagui.cfg — файл конфигурации для графического варианта программы. Файлы конфигурации должны располагаться в подкаталоге CFG главного каталога IDA Pro.

Загрузка исследуемого кода

При загрузке исполняемого модуля появляется окно, изображенное на рис. 5.2, при помощи которого можно настроить процесс загрузки и первичного анализа. Окно содержит большое количество настроек, которые мы разберем ниже. В абсолютном большинстве случаев IDA Pro предлагает оптимальную настройку, так что ничего настраивать не приходится, а остается только нажать кнопку **OK** и положиться на волю провидения и дизассемблер. Но поскольку эти опции все же иногда используются, я дам их краткое толкование.

- Список **Load file *каталог\имя* as** содержит перечень форматов, которые распознаются данной версией программы IDA Pro для выбранного модуля. В большинстве случаев IDA Pro сам распознает, какой тип файла предполагается загрузить. Кстати, в зависимости от этого типа автоматически устанавливаются и остальные опции окна. Вы можете, например, провести простой эксперимент и дизассемблировать DOS-заглушку PE-модуля *(см. рад. 1.5.1)*, выбрав в списке строку **MS-DOS executable**. Кнопка **Set** служит для фиксации выбора. Еще раз хочу подчеркнуть, что данный список соответствует тому, что мы выбрали PE-модуль. Действительно он может трактоваться и как обычный PE-модуль, и как MS-DOS-

программа, и как бинарный файл. В случае если выберете, скажем, NE-модуль, то содержимое списка будет другим.

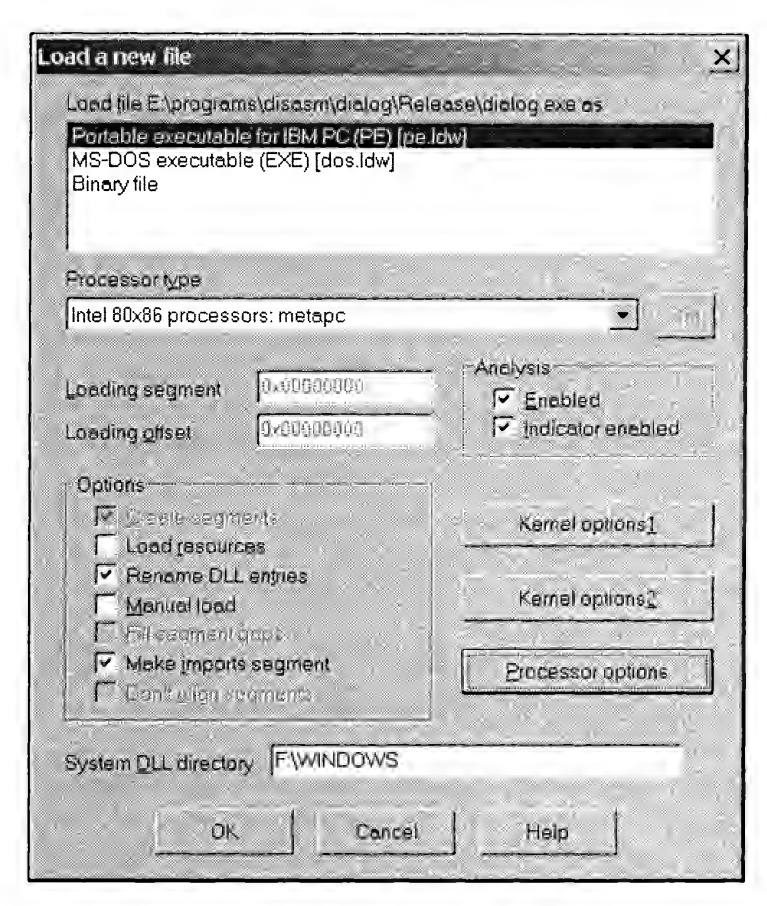


Рис. 5.2. Окно управления загрузкой исследуемого кода

- Выпадающий список **Processor type** предоставляет возможность выбрать процессор, для которого скомпилирован данный исполняемый модуль.
- Поля **Loading segment** и **Loading offset** позволяют загружать модуль в конкретный сегмент и с конкретным смещением, что может быть полезно для DOS- и двоичных модулей. Для PE-модулей эти параметры не используются.
- Флажок **Enabled** (группа **Analysis**) позволяет отменить первичный анализ исполняемого кода. По умолчанию флажок установлен, т. е. анализ после загрузки будет проводиться.

- Флажок **Indicator enabled** в установленном состоянии (по умолчанию) указывает проводить индикацию процесса анализа.
- Флажок **Create segments** (группа **Options**) для PE-модулей не используется. Если флажок установлен, то IDA Pro создает необходимые сегменты.
- Флажок **Load resources** предписывает загружать ресурсы PE-модуля. Для бинарного модуля флажок называется **Load as code segment** (Загрузить как кодовый сегмент) и используется, например, для com-программ[2].
- Если флажок **Rename DLL entries** не установлен, то IDA Pro делает дополнительные комментарии для функций, импортируемых по ординалу, в противном случае функции переименуются на усмотрение дизассемблера.
- Установка флажка **Manual load** предписывает дизассемблеру "советоваться" с вами на каждом шаге загрузки.
- Флажок **Fill segment gaps** актуален только для NE-модулей и предписывает заполнять пространство между сегментами, образуя один большой сегмент.
- Если установлен флажок **Make imports segments**, то заставляет дизассемблер трактовать секцию .idata, только в плане импортируемой информации, игнорируя тот факт, что в ней могут находиться и данные.
- Флажок **Don't align segments** предписывает выравнивать сегменты. Данный флажок не используется для рассматриваемых нами модулей.
- Кнопка **Kernel options1** вызывает окно опций, используемых при анализе исполняемого кода.
  - С помощью флажка **Create offsets and segments using fixup info** дизассемблеру предписывается использовать при анализе информацию из таблицы перемещений.
  - Флажок **Mark typical code sequence as code** предписывает дизассемблеру использовать при анализе типичные последовательности команд микропроцессора.
  - Флажок **Delete instructions with no xrefs** разрешает игнорировать инструкции микропроцессора, на которые нет перекрестных ссылок.
  - Флажок **Trace execution flow** позволяет проводить трассировку для обнаружения инструкций процессора.
  - Флажок **Create functions if call is present** предписывает распознавать функции по их вызовам.

---

[2] Com-программа — программа очень простого формата, используемого в операционной системе MS-DOS.

- Флажок **Analyse and create all xrefs** — одна из основных опций, которая заставляет дизассемблер использовать в своем анализе перекрестные ссылки.
- Флажок **Use flirt signatures** предписывает использовать технологию FLIRT (учет сигнатур для распознавания библиотечных функций).
- Флажок **Create function if data xref data->code32 exists** предписывает проверять ссылки на исполняемый код в области данных.
- Флажок **Rename jump functions as j_...** разрешает IDA Pro переименовывать простые функции, содержащие только команду перехода `jmp` *`somwhere`*, в `j_`*`somewhere`*.
- Флажок **Rename empty functions as nullsub_...** позволяет IDA Pro переименовывать функции, содержащие одну команду `RET`, в `nullsub_...`.
- Флажок **Create stack variables** предписывает дизассемблеру создавать (определять) локальные переменные и параметры функций.
- Флажок **Trace stack pointer** заставляет IDA Pro отслеживать значение регистра указателя стека ESP.
- Флажок **Create ascii string if data xref exists** предписывает рассматривать данное, на которое есть ссылка, как ASCII-строку, если его длина превосходит определенную величину.
- Флажок **Convert 32bit instruction operand to offset** предписывает рассматривать непосредственное данное в инструкции процессора, как адрес, если его значение попадает в определенный промежуток.
- Флажок **Create offset if data xref to seg32 exists** предписывает рассматривать значение, хранящееся в области данных, как адрес, если оно попадает в определенный промежуток.
- Флажок **Make final analysis pass** сообщает, что дизассемблеру на последней стадии анализа следует преобразовывать все еще неисследованные байты в данные или инструкции.

❒ Кнопка **Kernel options2** вызывает окно с еще одним набором опций, используемых при анализе исполняемого кода.

- Флажок **Locate and create jump tables** предписывает IDA Pro делать заключение об адресе и размере таблицы переходов.
- Если флажок **Coagulate data in the final pass** сброшен, то на последней стадии анализа преобразуются только байты сегмента кода (см. флажок **Make final analysis pass**).
- Флажок **Automatically hide library functions** заставляет скрывать (сворачивать) библиотечные функции, обнаруженные при помощи технологии FLIRT.

* Флажок **Propagate stack argument information** указывает дизассемблеру сохранять информацию о стековых параметрах вызова при последующих вызовах (вызов функции из другой функции и т. д.).
* Флажок **Propagate register argument information** указывает дизассемблеру сохранять информацию о регистровых параметрах вызова при последующих вызовах (вызов функции из другой функции и т. д.).
* Флажок **Check for Unicode strings** разрешает дизассемблеру осуществлять проверку программы на наличие строк в кодировке Unicode.
* Флажок **Comment anonymous library functions** указывает, что дизассемблер должен помечать анонимные библиотечные функции с использованием имени библиотеки и сигнатуры, с помощью которой функция была обнаружена.
* Флажок **Multiple copy library function recognition** разрешает дизассемблеру распознавать в программе несколько копий одной и той же функции.
* Флажок **Create function tails** разрешает поиск и добавление к определению функции ее окончания.

- Кнопка **Processor options** вызывает окно с опциями процессора.
  * Флажок **Convert immediate operand of "push" to offset** указывает на возможность преобразовывать непосредственный операнд в команде `PUSH` в смещение (адрес).
  * Флажок **Convert db 90h after "jmp" to "nop"** указывает дизассемблеру, что байты `90H`, стоящие после команды `JMP`, следует трактовать как команды `NOP`.
  * Флажок **Convert immediate operand of "mov reg,..." to offset** указывает на возможность преобразовывать непосредственный операнд в команде `MOV` *`reg,...`* (*`reg`* — регистр) в смещение (адрес).
  * Флажок **Convert immediate operand of "mov memory,..." to offset** предполагает возможность преобразовывать непосредственный операнд в команде `MOV mem,...` в смещение (адрес).
  * Флажок **Disassemble zero opcode instructions** предписывает дизассемблировать следующую инструкцию: `00 00 ADD [EAX],AL`. По умолчанию данный флажок сброшен.
  * Флажок **Advanced analysis of Borland's RTTI** (RTTI, runtime type identification, идентификация типов во время исполнения программы) разрешает IDA Pro проверять и создавать структуры RTTI.
  * Флажок **Check 'unknown_libname' for Borland's RTTI** позволяет проверять имена, помеченные как "unknown_libname" на наличие RTTI-структуры.

- Флажок **Advanced analysis of catch/finally block after function** разрешает дизассемблеру искать `catch/finally` блоки обработки исключений.
- Флажок **Allow references with different segment bases** разрешает дизассемблеру указывать ссылки на символы, даже если величина, хранящаяся по указанному адресу, символом не является (не является кодом символа).
- Флажок **Don't display redundant instruction prefixes** предписывает дизассемблеру не показывать некоторые префиксы команд для улучшения читаемости листинга.
- Флажок **Interpret int 20 as VxDcall** предписывает интерпретировать `INT 20H` как `VxDcall/jump`.
- Флажок **Enable FPU[3] emulation instructions** указывает, что команды типа `INT 3?H` следует интерпретировать как эмуляцию команд арифметического сопроцессора.
- Если флажок **Explicit RIP-addressing** установлен, то предполагается, что в программе используется RIP-адресация (Relative Instruction Pointer, относительный указатель команды). Данный флаг действует для 64-битных процессоров.

❒ Поле **System DLL directory** содержит каталог, где IDA Pro будет искать динамические библиотеки, если в подкаталоге IDS отсутствует файл с расширением ids, соответствующий данной библиотеки.

Окно дизассемблера

Поскольку работать в IDA Pro большей частью приходится в окне дизассемблера, то есть смысл остановиться на нем более подробно. Надо сказать, что разработчики дизассемблера действительно тщательно продумали представление дизассемблированной информации и способы навигации по ней. Мы остановимся только на некоторых ключевых моментах.

❒ **Сворачивание функций.** Функции в окне дизассемблера могут представляться в свернутом, или скрытом виде (hide), и развернутом, или раскрытом (unhide) виде. В свернутом виде функция представляется всего одной строкой. Это действительно замечательное изобретение позволяет значительно улучшить читаемость дизассемблированного текста. Скрытие и раскрытие функции осуществляется клавишами <-> и <+> (на дополнительной клавиатуре) или с использованием команд **View | Hide** и **View | Unhide**.

---

[3] FPU (Floating Point Unit), т. е. дословно "устройство с плавающей точкой". Так изначально называли арифметический сопроцессор. Другой английский синоним NPU — Numeric Processor Unit.

- **Индикация переходов.** На рис. 5.3 представлено окно дизассемблера. Обратите внимание на крайнюю левую секцию окна. Она предназначена для улучшения ориентации пользователя в листинге. Точками обозначены команды. Строка без точки означает, что здесь расположен комментарий. Щелчок по точке приводит к установке точки останова на данный адрес. Переходы помечаются сплошными и пунктирными линиями. Сплошные линии обозначают безусловные переходы, пунктирные линии — условные переходы.

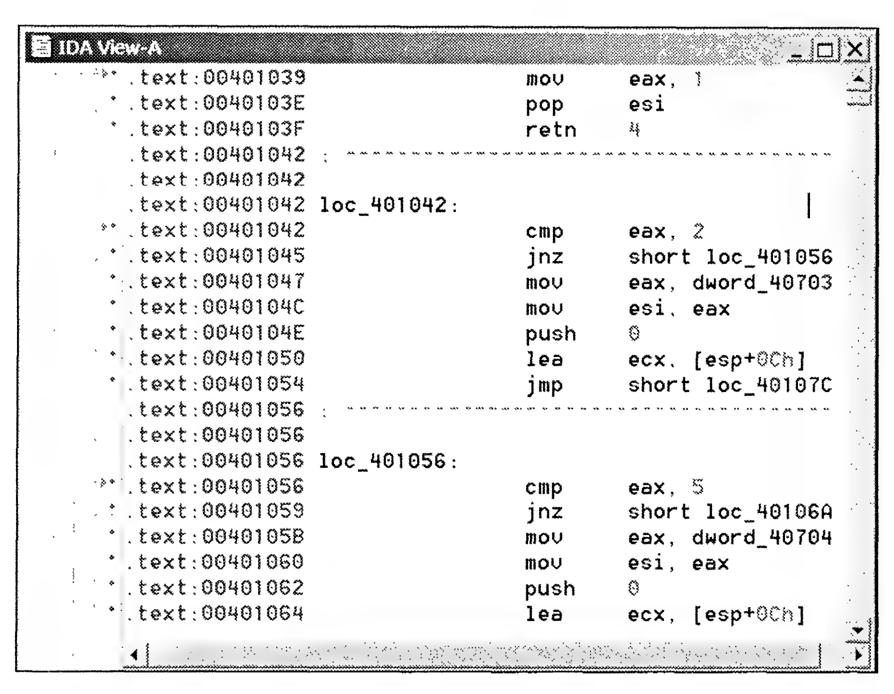


Рис. 5.3. Индикация переходов в окне дизассемблера

- **Комментарии.** Адрес в программе, куда осуществляется переход (команды условных и безусловных переходов или команда CALL) или на который имеется ссылка, содержит специальный комментарий. Комментарий начинается либо с CODE XREF, если ссылка имеет смысл перехода на данный адрес, либо с DATA XREF, если на эту инструкцию ссылаются как на данные (например, так MOV EAX,OFFSET L1). Это есть не что иное, как *перекрестные ссылки*. Перекрестные потому, что данный адрес и есть тот перекресток, где встречаются ссылки из других мест программы. Далее через двоеточие указывается адрес относительно начала функции или начала секции, откуда идет эта ссылка. Наведя курсором мыши на этот адрес, мы вызовем всплывающее окно с фрагментом кода, откуда ссылают-

ся на данную инструкцию. Адрес обязательно содержит символы ↑, ↓, показывающие направление, где находится строка, ссылающая на данную строку. Перейти на строку, откуда идет ссылка, можно просто двойным щелчком мыши по адресу. Если ссылок на данную строку меньше четырех, то они перечисляются, в противном случае указывается многоточие. В этом случае можно щелкнуть по одному из адресов правой кнопкой мыши и выбрать пункт контекстного меню **Jump to cross reference**. После этого появится окно со списком всех адресов, где имеется ссылка на данную строку. Выбрав нужный адрес щелчком мыши (либо нажатием кнопки **OK**), мы окажемся в нужном месте листинга. На рис. 5.4 представлен фрагмент окна дизассемблера, содержащего справа комментарии с указанием перекрестных ссылок.

```
.text:004028F1
.text:004028F1 loc_4028F1:                                ; DATA XREF: .rdata:stru_4075F0↓o
.text:004028F1                 mov     esp, [ebp-18h]
.text:004028F4
.text:004028F4 loc_4028F4:                                ; CODE XREF: sub_4028C0+27↑j
.text:004028F4                                            ; sub_4028C0+2B↑j
.text:004028F4                 or      dword ptr [ebp-4], 0FFFFFFFFh
.text:004028F8                 add     dword ptr [ebp-1Ch], 4
.text:004028FC                 jmp     short loc_4028D3
.text:004028FE ; ---------------------------------------------------------------------------
.text:004028FE
.text:004028FE loc_4028FE:                                ; CODE XREF: sub_4028C0+1A↑j
```

Рис. 5.4. Перекрестные ссылки

- **Обозначение адреса.** В листинге, который представлен в окне дизассемблера, используются различные способы обозначения адреса. Например, в случае библиотечной функции или функции API явно указывается имя этой функции. Кроме этого, IDA Pro практикует называть ссылки на обнаруженные им строки, на основе содержимого строки. Например, если строка содержит текст "You are wrong!", то IDA Pro обозначает ссылку на эту строку, как `aYouAreWrong`. Префикс в данном случае указывает, что IDA Pro считает это ASCII-строкой. Остальные имена, обозначающие имена функций или адреса данных, формируются на основе префикса и адреса. Вы можете встретить следующие префиксы:
  - `sub_` — для обозначения функции;
  - `locret_` — адрес инструкции `return`;
  - `loc_` — адрес инструкции;
  - `off_` — данные, содержащие адрес (смещение);
  - `seg_` — данные, содержащие сегментный адрес;
  - `asc_` — адрес ASCII-строки;

- byte_ — адрес байта;
- word_ — адрес слова;
- dword_ — адрес двойного слова;
- qword_ — адрес 64-битной величины;
- flt_ — адрес 32-битного вещественного числа;
- dbl_ — адрес 64-битного вещественного числа;
- tbyte_ — адрес 80-битного вещественного числа;
- stru_ — адрес структуры;
- algn_ — директива выравнивания;
- unk_ — адрес неисследованной области.

- **Контекстное меню.** При работе с окном дизассемблера удобно пользоваться контекстным меню, которое появляется, если щелкнуть в окне правой кнопкой мыши. При этом меню отличается некоторыми пунктами для разных частей листинга: названий функций, инструкций, комментариев, выделенного блока и т. д. Часть пунктов касается работы IDA Pro в качестве отладчика, и мы об этом еще будем говорить (**Run to cursor**, **Add breakpoint**, **Add execution trace**). В частности, обратите внимание на пункт **Rename**, с помощью которого, например, можно редактировать содержимое команд (операнды).
- **Навигация по листингу.** Наконец, важным моментом является передвижение по листингу. Выход на перекрестную ссылку мы уже разбирали. Но точно также (двойным щелчком мыши) можно двигаться в обратную сторону по перекрестной ссылке (например, безусловному переходу, команде CALL или по адресу в команде типа MOV EAX,OFFSET *addres*). Причем IDA Pro запоминает все ваши переходы. Так, вы всегда можете двигать назад по цепочке или опять по ней же вперед (в стиле интернет-браузера), используя кнопки на панели инструментов.

Другие окна

- Окно **Hex View**. Содержит шестнадцатеричный дамп загруженного модуля, а также ASCII-символы, соответствующие этому дампу. Окно является вспомогательным по отношению к окну дизассемблера и легко синхронизируется с ним, для чего достаточно щелкнуть правой кнопкой мыши в окне и выбрать пункт **Synchronize with | Ida View** контекстного меню. Перейдя в окно дизассемблера, мы окажемся как раз в том месте программы, которое соответствует адресу в окне дампа. Кроме этого, IDA Pro отслеживает адреса, с которыми работает окно дизассемблера, и при переходе к дампу мы оказываемся в нужном месте.

- Окно **Exports**. Содержит список экспортируемых функций. Актуально для динамических библиотек. Для обычных исполняемых модулей список состоит из одного элемента — функции `start`.
- Окно **Imports**. Содержит список импортируемых функций и модули, откуда они импортируются. Двойной щелчок по импортируемой функции приводит к переходу в окно дизассемблера, в точку входа. Так что мы далее легко можем найти все перекрестные ссылки на эту функцию в программе.
- Окно **Names**. Окно имен содержит список не только всех импортируемых или библиотечных функций, но и распознанные IDA Pro имена переменных и меток. Слева для каждого имени стоит значок, определяющий тип имени:
  - **L** — библиотечная функция;
  - **F** — регулярные функции, API-функции;
  - **C** — инструкция (метка);
  - **A** — строка в кодировке ASCII;
  - **D** — данные;
  - **I** — импортируемая функция.

  Двойной щелчок по имени приводит к переходу к тому месту программы, где это имя используется. С помощью клавиши <Insert> можно создать новое имя (например, для метки) и указать адрес, соответствующий этому имени. Имя, разумеется, появится и в окне дизассемблера.
- Окно **Functions**. Данное окно содержит весь перечень определенных IDA Pro функций (и библиотечных, и импортируемых, и пользовательских).
- Окно **Strings**. Содержит все найденные дизассемблером строки. Сделав двойной щелчок по строке, мы автоматически перейдем в текст листинга, где эта строка определена. По умолчанию в окне представляются только строки в стиле C. Щелкнув по окну правой кнопкой мыши и выбрав пункт **Setup** в контекстном меню, мы можем включить в окно строки другой структуры (строки Unicode, строки, используемые в языке Pascal, и т. д.).
- Окно **Structures**. Содержит все найденные дизассемблером структуры. При помощи клавиши <Insert> можно добавить в список новую структуру.
- Окно **Enums**. Служит для указания найденных в программе перечислений (enumerations).

Кроме перечисленных окон при работе дизассемблера будут использоваться и другие. В частности, обращаю ваше внимание на окно **Libraries**. В справочной системе это окно называется *окном сигнатур*. В окне содержится перечень сигнатур, которые были использованы при распознавании библио-

течных функций. На рис. 5.5 представлено окно сигнатур. Как видно из рисунка, в списке указано имя файла, где содержатся сигнатуры функций, количество найденных с помощью этих сигнатур функций, а также имя библиотеки, к функциям которой были применены данные сигнатуры. Нажав клавишу <Insert>, вы можете выбрать из появившегося списка файлов сигнатур нужный файл, сигнатуры которого тут же будут использованы для распознавания новых функций, и добавить его.

List of applied library modules

File	State	#func	Library name
bh32rw32	Applied	230	BCC v4.x/5.x & Builder v1.0/v6.0 win32 runtime
b32vcl	Applied	10	Borland Visual Component Library & Packages

Рис. 5.5. Окно сигнатур

Меню и панели инструментов

Я не собираюсь подробно разъяснять все пункты меню и все кнопки панели инструментов IDA Pro. В большей части возможностей IDA Pro читатель легко разберется сам. Хочется остановиться на некоторых, с моей точки зрения, важных моментах.

- Меню **File** содержит следующие пункты:
  - **Open** — загрузка дизассемблируемого модуля;
  - **Load** — загрузка различных файлов: **reload the input file** — повторная загрузка дизассемблируемого модуля; **Additional binary file** — загрузка в базу дополнительного бинарного файла; **IDS file** — загрузка IDS-файла, содержащего информацию о функциях той или иной библиотеки импорта, все IDS-файлы, находящиеся в каталоге IDS, загружаются автоматически; **PDB file** — загрузка PDB-файла, который содержит отладочную информацию; **DBG file** — загрузка файла, содержащего отладочную информацию; **FLIRT signature file** — загрузка и применение файла сигнатур, такая же операция выполняется в окне сигнатур (см. рис. 5.5 и комментарий к нему); **Parse C header file** — чтение из

заголовочного файла C определений типов для дальнейшего объявления новых структур и перечислений (см. описание окон **Enums** и **Structures** *в предыдущем разделе*);

- **Produce file** — при помощи данного пункта можно создавать новые файлы различной структуры на основе дизассемблируемого кода: MAP-файл, который может быть использован отладчиками, файл на языке ассемблера (расширение asm), LST-файл (листинг), листинг в формате HTML и др.;
- **IDC file** — загрузка и выполнение файла сценария (скрипта) *(см. разд. 5.1.3)*;
- **IDC command** — вызов окна немедленного выполнения скриптов;
- **Save** — сохранение текущей базы дизассемблирования (файл с расширением idb);
- **Save as** — сохранение текущей базы дизассемблирования с заданным именем;
- **Close** — закрытие дизассемблируемого файла с сохранением базы дизассемблирования.

❑ Меню **Edit** содержит следующие пункты:

- **Copy** — копировать в буфер обмена выделенный фрагмент;
- **CODE** — преобразовать блок к исполняемому коду;
- **DATA** — преобразовать блок к данным;
- **Struct var** — преобразовать блок к выбранной структуре;
- **Strings** — преобразовать к строке. Типы строк предлагаются в подменю;
- **Array** — преобразовать к массиву с заданными параметрами;
- **Undefine** — отметить как данные неопределенной структуры;
- **Name** — переименовать;
- **Operand type** — задать тип операнда;
- **Comments** — управление комментарием;
- **Segments** — управление сегментами;
- **Structs** — управление структурами;
- **Functions** — управление функциями;
- **Other** — другие возможности: указать директиву выравнивания, ввести инструкцию или данные, выделить цветом;
- **Plugins** — плагины (внешние подключаемые модули, от англ. *plug in* — подключать).

- Меню **Jump**. Пункты посвящены всевозможным переходам по дизассемблированному тексту: переход по указанному адресу, переход к указанной функции (выбор функции из списка), переход к точке входа программы, пометка строки, переход по указанной метке и т. д.
- Меню **Search**. Пункты реализуют различные операции поиска в дизассемблированном тексте: поиск текста, поиск следующего в тексте блока данных, поиск следующей в тексте ассемблерной инструкции, поиск последовательности байтов и т. д.
- Меню **View**. С помощью пунктов данного меню можно менять внешний вид дизассемблера IDA Pro: открывать новые окна (**Open Subviews**), создавать и удалять панели инструментов (**Toolbars**), сворачивать (**Hide**) и разворачивать (**Unhide**) функцию, вызвать окно калькулятора и др.
- Меню **Debugger**. Пункты меню относятся к отладочным возможностям IDA Pro: управление точками прерывания (**Breakpoints**), управление наблюдениями (**Watches**), управление трассировкой (**Tracing**), просмотр содержимого регистров (**General registers**, **Segment registers**, **FPU registers**) и др.
- Меню **Options**. Пункты посвящены всевозможным настройкам IDA Pro, с частью из которых мы знакомились, когда рассматривали окно управления загрузкой.
- Меню **Windows**. С помощью пунктов данного меню можно управлять окнами IDA Pro.
- Меню **Help**. Пункты посвящены получению справки и технической поддержки.

Ключи запуска программы

При запуске IDA Pro можно использовать следующие ключи:

- `-a` — отмена автоматического анализа;
- `-A` — запуск IDA Pro с автоматической загрузкой последней базы;
- `-b####` — указание адреса для загрузки модуля;
- `-B` — запуск IDA Pro с автоматической генерацией idb- и asm-файлов;
- `-c` — удаление старой базы дизассемблирования;
- `-ddirectiva` — запуск с указанием директивы загрузки для первого прохода анализа;
- `-Ddirective` — запуск с указанием директивы загрузки для второго прохода анализа;
- `-f` — исключение инструкций арифметического сопроцессора;
- `-h` — открытие окна помощи IDA Pro;

- `-i` — указание адреса точки входа в программу;
- `-M` — запрет использования мыши (для консольного варианта загрузки);
- `-O####` — передача опций дополнительному модулю (plugin):

  `-Oplug1:opt1:opt2:opt3`

  Здесь `plug1` — название дополнительного модуля; `opt1`, `opt2`, `opt3` — опции модуля;
- `-o####` — указание имени базы. Ключ используется вместе с ключом `-c`;
- `-p` — указание типа процессора;
- `-P+` — упаковка базы;
- `-P-` — отказ от упаковки базы;
- `-R` — загрузка ресурсов из exe-файла;
- `-S####` — выполнение указанного idc-файла;
- `-W####` — указание каталога Windows;
- `-x` — отказ от создания сегментов;
- `-?` — экран помощи о ключах запуска IDA Pro.

5.1.2. Простые примеры исследования кода

В данном разделе мы опять вернемся к примерам на языке ассемблера *(см. разд. 1.6)*. Причина очень проста. С помощью языка ассемблера очень легко смоделировать нужную программную ситуацию, чтобы продемонстрировать те или иные закономерности исследования кода в дизассемблере IDA Pro.

О возможностях IDA Pro

Простые вещи

В предыдущих главах мы неоднократно убеждались в возможностях дизассемблера IDA Pro анализировать исполняемый код. Рассмотрим простую программу на языке ассемблера из листинга 5.1.

Листинг 5.1

```
.586P
.MODEL FLAT,STDCALL
includelib e:\masm32\lib\user32.lib
EXTERN  MessageBoxA@16:NEAR
;сегмент данных
```

```
_DATA SEGMENT
TEXT1 DB 'No problem!',0
TEXT2 DB 'Message',0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
  PUSH OFFSET 0
  PUSH OFFSET TEXT2
  PUSH OFFSET TEXT1
  PUSH 0
  CALL MessageBoxA@16
  MOV  ESI,3
  ADD  ESI,OFFSET L2
L2:
  CALL ESI
  RETN
L1:
  XOR  EAX,EAX
  RETN
_TEXT ENDS
END START
```

Конечно, от читателя не утаить маленькую хитрость, которая таится в строках

```
MOV  ESI,3
ADD  ESI,OFFSET L2
L2:
CALL ESI
RETN
L1:
```

Команда `CALL ESI` осуществляет переход на метку `L1`. Как на это среагирует IDA Pro? Рассмотрим результирующий листинг 5.2.

Листинг 5.2

```
.text:00401000  _text   segment para public 'CODE' use32
.text:00401000          assume cs:_text
```

```
.text:00401000          ;org 401000h
.text:00401000    assume es:nothing, ss:nothing, ds:_data, fs:nothing,
gs:nothing
.text:00401000 ;-------- S U B R O U T I N E ------------
.text:00401000      public start
.text:00401000    start            proc near
.text:00401000      push    0               ; uType
.text:00401002      push    offset Caption  ; lpCaption
.text:00401007      push    offset Text     ; lpText
.text:0040100C      push    0               ; hWnd
.text:0040100E      call    MessageBoxA
.text:00401013      mov     esi, 3
.text:00401018      add     esi, offset loc_40101E
.text:0040101E  loc_40101E:      ; DATA XREF: start+18?o
.text:0040101E      call    esi  ; sub_401021
.text:00401020      retn
.text:00401020  start    endp
.text:00401021    ; --------- S U B R O U T I N E -----------
.text:00401021
.text:00401021  sub_401021  proc near  ; CODE XREF: start:loc_40101E?p
.text:00401021      xor     eax, eax
.text:00401023      retn
.text:00401023  sub_401021  endp
.text:00401023
.text:00401024 ; [00000006 BYTES: COLLAPSED FUNCTION MessageBoxA. PRESS
KEYPAD "+" TO EXPAND]
.text:0040102A      align 200h
.text:0040102A  _text    ends
```

Смотрите, IDA Pro четко отследил значение регистра ESI и тем самым определил начало процедуры sub_401021. Конечно, арифметика здесь проста: нужно добавить к адресу loc_40101E число три, и получим как раз адрес вызываемой процедуры. Определив же начало процедуры, можно достаточно легко выяснить ее конец, который в данном случае определяется просто ближайшей к началу командой возврата RETN.

Несколько видоизменим программу из листинга 5.1. Оказывается, наше незначительное изменение приводит к некоторым сложностям в дизассемблировании (листинг 5.3).

Листинг 5.3

```
.586P
.MODEL FLAT,STDCALL
includelib e:\masm32\lib\user32.lib
EXTERN  MessageBoxA@16:NEAR
;сегмент данных
_DATA SEGMENT
TEXT1 DB 'No problem!',0
TEXT2 DB 'Message',0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
  PUSH OFFSET 0
  PUSH OFFSET TEXT2
  PUSH OFFSET TEXT1
  PUSH 0
  CALL MessageBoxA@16
  MOV  ESI,3
  ADD  ESI,OFFSET L2
  PUSH ESI
  POP  EDI
L2:
  CALL EDI
  RETN
L1:
  XOR  EAX,EAX
  RETN
_TEXT ENDS
END START
```

Рассмотрим, что получилось у дизассемблера IDA Pro после анализа исполняемого кода, созданного транслятором ассемблера из программы, которую мы видим в листинге 5.3. Результат представлен в листинге 5.4.

Листинг 5.4

```
.text:00401000  _text      segment para public 'CODE' use32
.text:00401000        assume cs:_text
```

```
.text:00401000        ;org 401000h
.text:00401000  assume es:nothing, ss:nothing, ds:_data, fs:nothing,
gs:nothing
.text:00401000 ;----------- S U B R O U T I N E-------------
.text:00401000      public start
.text:00401000  start    proc near
.text:00401000      push    0               ; uType
.text:00401002      push    offset Caption  ; lpCaption
.text:00401007      push    offset Text     ; lpText
.text:0040100C      push    0               ; hWnd
.text:0040100E      call    MessageBoxA
.text:00401013      mov     esi, 3
.text:00401018      add     esi, offset loc_401020
.text:0040101E      push    esi
.text:0040101F      pop     edi
.text:00401020  loc_401020:        ; DATA XREF: start+18?o
.text:00401020      call    edi
.text:00401022      retn
.text:00401022  start    endp
.text:00401023  ;--------------------------------------------------
.text:00401023      xor     eax, eax
.text:00401025      retn
.text:00401026  ; [00000006 BYTES: COLLAPSED FUNCTION MessageBoxA. PRESS
KEYPAD "+" TO EXPAND]
.text:0040102C      align 200h
.text:0040102C  _text    ends
```

Рассмотрим листинг 5.4, где представлен анализ, который провел дизассемблер IDA Pro. Посмотрите, незначительные изменения в исходном тексте программы приводят к тому, что процедура по адресу `00401023` более не распознается дизассемблером. Конечно, чтобы все-таки понять, как проводился анализ, надо посмотреть алгоритм, которым пользуется IDA Pro. Но некоторые соображения можно сделать, и не видя алгоритма. Как я уже говорил, IDA Pro — очень осторожная программа. Она не хочет делать слишком скороспелые выводы. В данном случае есть некоторая вероятность, что на метку `loc_401020` будет сделан переход (неявный, конечно) откуда-то из другого места программы, и тогда, возможно, адрес процедуры будет совсем иным. Конечно, все это трудно взвесить, но из осторожности можно учесть такую возможность и положиться на совместную работу с пользователем.

Впрочем, вот такой фрагмент

```
PUSH ESI
POP  ESI
L2:
CALL ESI
```

IDA Pro уже нисколько не смущает, и адрес процедуры в этом случае определяется вполне корректно.

Как это происходит

Давайте рассмотрим, как происходит совместная работа исследователя кода и дизассемблера IDA Pro. В листинге 5.5 представлена простая программа на ассемблере. Вызов `CALL EDI`, как легко видеть, осуществляется по адресу, соответствующему метке `L1`.

Листинг 5.5

```
.586P
.MODEL FLAT,STDCALL
includelib e:\masm32\lib\user32.lib
EXTERN  MessageBoxA@16:NEAR
;сегмент данных
_DATA SEGMENT
TEXT1 DB 'No problem!',0
TEXT2 DB 'Message',0
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
  MOV  ESI,3
  PUSH ESI
  PUSH OFFSET 0
  PUSH OFFSET TEXT2
  PUSH OFFSET TEXT1
  PUSH 0
  CALL MessageBoxA@16
  POP  EDI
  ADD  EDI,OFFSET L2
```

```
L2:
  CALL EDI
  RETN
L1:
  XOR  EAX,EAX
  RETN
_TEXT ENDS
END START
```

Оттранслируем программу, а потом загрузим код в дизассемблер IDA Pro. И вот, что у нас должно получиться (листинг 5.6).

Листинг 5.6

```
.text:00401000  _text    segment para public 'CODE' use32
.text:00401000      assume cs:_text
.text:00401000      ;org 401000h
.text:00401000  assume es:nothing, ss:nothing, ds:_data, fs:nothing,
gs:nothing
.text:00401000 ; --------------- S U B R O U T I N E ------------------
.text:00401000
.text:00401000      public start
.text:00401000  start    proc near
.text:00401000      mov     esi, 3
.text:00401005      push    esi
.text:00401006      push    0               ; uType
.text:00401008      push    offset Caption  ; lpCaption
.text:0040100D      push    offset Text     ; lpText
.text:00401012      push    0               ; hWnd
.text:00401014      call    MessageBoxA
.text:00401019      pop     edi
.text:0040101A      add     edi, offset loc_401020
.text:00401020
.text:00401020  loc_401020:       ; DATA XREF: start+1A?o
.text:00401020      call    edi
.text:00401022      retn
.text:00401022  start    endp
.text:00401023      xor     eax, eax
.text:00401025      retn
```

```
.text:00401026    ; [00000006 BYTES: COLLAPSED FUNCTION MessageBoxA.
PRESS KEYPAD "+" TO EXPAND]
.text:0040102C      align 200h
.text:0040102C  _text    ends
```

Как и следовало ожидать, дизассемблер не распознает адрес, по которому будет осуществлен вызов CALL EDI.

Начнем наши изыскания с того, что создадим функцию по адресу 00401023. То, что мы имеем здесь дело с некоторой функцией, ясно даже без определения адреса, по которому осуществляется вызов CALL EDI. Действительно, последовательность команд

```
XOR EAX,EAX
RETN
```

почти наверняка указывает на тело некоторой функции. Установим курсор на первую команду предполагаемой функции и нажмем клавишу <P> или воспользуемся пунктом меню **Edit | Functions | Create function**. В результате IDA Pro автоматически создаст функцию:

```
.text:00401023 sub_401023      proc near
.text:00401023                 xor     eax, eax
.text:00401025                 retn
.text:00401025 sub_401023      endp
```

Итак, функция создана, и теперь ссылки на нее можно использовать в дизассемблированном тексте. При этом дизассемблер будет автоматически учитывать наше редактирование и продолжать анализ с учетом нашей корректировки. Перейдем теперь к строке с адресом 00401020 (CALL EDI). Нажмем клавишу <;> для ввода комментария. Можно воспользоваться также пунктом меню: **Edit | Comments Enter comments**. В результате на экране появится окно для ввода комментария (рис. 5.6). Здесь можно ввести любой комментарий. Но комментарий в IDA Pro обладает одной особенностью: некоторые комментарии несут информацию не только для нас с вами, но для самого дизассемблера.

Итак, в окно редактирования введем следующую строку: DATA XREF: sub_401023. Мы, таким образом, указываем, что вызов процедуры осуществляется по адресу, соответствующему метке sub_401023. Результат более чем интересен. Мы не просто получаем комментарий, щелкнув по которому, переходим по соответствующей ссылке. Автоматически появляется комментарий и у строки с адресом 0040101. Чтобы не быть голословным, привожу следующий фрагмент (листинг 5.7).

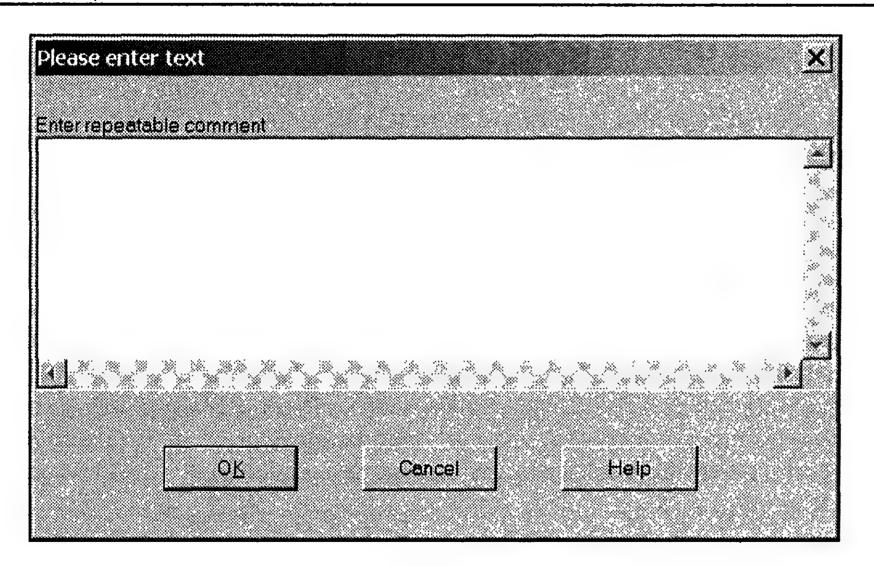


Рис. 5.6. Ввод комментария

Листинг 5.7

```
.text:00401019    pop     edi
.text:0040101A    add   edi, offset loc_401020 ; DATA XREF: sub_401023
.text:00401020  loc_401020:        ; DATA XREF: start+1A?o
.text:00401020    call    edi      ; DATA XREF: sub_401023
.text:00401022    retn
.text:00401022  start            endp
.text:00401023 ;--------------- S U B R O U T I N E---------------
.text:00401023
.text:00401023  sub_401023  proc near
.text:00401023    xor  eax, eax
.text:00401025    retn
.text:00401025  sub_401023  endp
```

Отладка в среде IDA Pro

Хотя отладка — не прямая обязанность IDA Pro, эта функция вполне работоспособна, и на ней стоит остановиться отдельно.

После загрузки исполняемого модуля в дизассемблер IDA Pro можно запустить отладчик. Однако прежде следует определиться с первой точкой останова. Проще всего использовать команду выполнения до текущего положения курсора, воспользовавшись пунктом меню **Debugger | Run to cursor** или

нажав клавишу <F4>. Естественно в качестве точки первого останова[4] выбрать первую инструкцию функций `main` или `WinMain`, а далее можно осуществить пошаговое выполнение с заходом (клавиша <F7>) или без захода (клавиша <F8>) в функции. Но можно также воспользоваться командой запуска процесса (пункт меню **Debugger | Start process** или клавиша <F9>), предварительно установив точку (точки) останова. Устанавливать точки останова можно прямо в дизассемблированном тексте при помощи клавиши <F2>. При этом строка, где находится инструкция, окрасится другим (красным по умолчанию) цветом. Наконец, можно воспользоваться окном настройки отладчика (пункт меню **Debugger | Debugger options**), которое представлено на рис. 5.7. В группе **Events** флажки определяют события, на которые реагирует отладчик, можно отметить флажок **Stop on debugging start** (Отладчик останавливается в момент своего запуска) или флажок **Stop on process entry points** (Отладчик останавливается на первой исполняемой инструкции программ[5]).

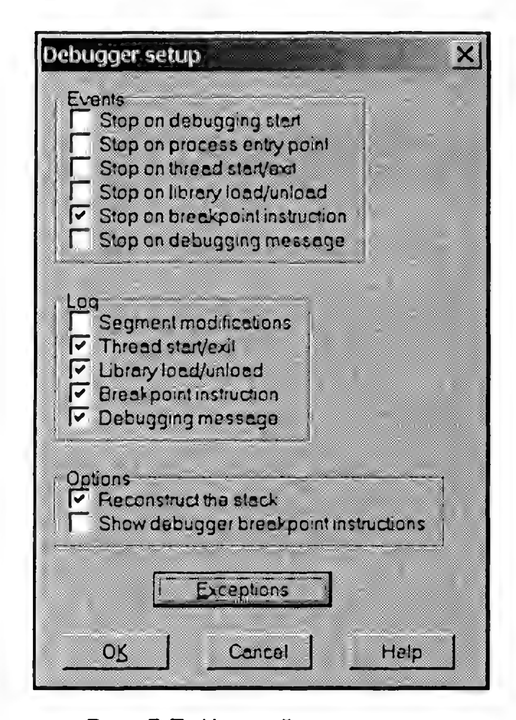


Рис. 5.7. Настройка отладчика

[4] Такую точку останова мы ранее назвали одноразовой *(см. разд. 4.2.2)*.

[5] Надеюсь, вы понимаете, что, скорее всего, эта инструкция не совпадет с первой инструкцией функции `main` или `WinMain`

Замечание

Флажок **Stop on thread start/exit** меня несколько удивляет. Действительно, thread — это поток. Но при создании процесса всегда создается, по крайней мере, один (его называют главным) поток. Но разработчики этот момент, очевидно, проигнорировали, имея в виду только потоки, явно создаваемые в программе.

Итак, с первой точкой останова мы определились. Что же мы имеем в своем арсенале, когда используем IDA Pro как отладчик. Вот некоторые ключевые моменты.

- После запуска приложения внешний вид IDA Pro изменяется. Появляются отладочные окна, при помощи которых можно контролировать процесс отладки. Это окно **View EIP**, содержащее отлаживаемый текст, окно **View ESP**, где представлено содержимое стека и текущее значение `ESP`, окно **General registers**, отражающее текущие значения общих регистров и регистра флагов, и окно **Threads** с информацией о потоках приложения. Отладчик всегда показывает в списке тот поток, где происходят отладочные события. Кроме этого, дополнительно можно открыть окно **FPU registers** с содержимым регистров сопроцессора (окно открывается автоматически, если выполняются инструкции арифметического сопроцессора) и окно **Modules**, где дан список загруженных модулей.
- Можно выполнять пошаговое выполнение программы: **Debugger | Step over** (клавиша <F8>) и **Debugger | Step into** (клавиша <F7>), выполнять до первой попавшейся команды `return` (комбинация клавиш <Ctrl>+<F7>), приостанавливать выполнение приложения (**Debugger | Pause process**).
- Можно наблюдать за указанными ячейками памяти (окно **Watch list**, пункт меню **Debugger | Watchs | Watch list**), задав их, используя пункт меню **Debugger | Watchs | Add watch**.
- Можно использовать *трассировку*, т. е. запись состояния отлаживаемого приложения на каждом шаге отладки. Для управления трассировкой предназначено подменю **Debugger | Tracing**. Все трассировочные события отображаются в окне трассировки **Trace window**. Можно отображать следующие трассировочные события: выполнение инструкции, выполнение функции, операцию записи в память, операции чтения/записи, выполнение инструкции.

5.2. Встроенный язык IDA Pro

Дизассемблер IDA Pro имеет встроенный язык программирования, с помощью которого можно писать небольшие программы анализа дизассемблированного кода, расширяя функциональность дизассемблера.

5.2.1. О встроенном языке IDA Pro

Встроенный язык IDA Pro — сильно упрощенный язык С. Этот язык называется сокращенно IDC (от англ. *Interactive Disassembler C*). Подкаталог IDC содержит несколько программ, написанных на этом языке, которые IDA Pro использует при анализе дизассемблированных текстов. Программы легко анализируются, так что на них можно изучать данный язык.

Общие сведения

Имеются два способа выполнять команды языка IDC.

Первый способ заключается в использовании командного окна. Вызвать командное окно можно из меню **File | IDC command** либо комбинацией клавиш <Shift>+<F2>. Внешний вид окна изображен на рис. 5.8. В поле редактирования вы можете вводить последовательность отделенных друг от друга точкой с запятой команд языка IDC. После нажатия кнопки **OK** IDA Pro будет пытаться интерпретировать записанные команды и выполнить их. Таким образом, используя данное окно, можно писать простейшие программы на языке IDC.

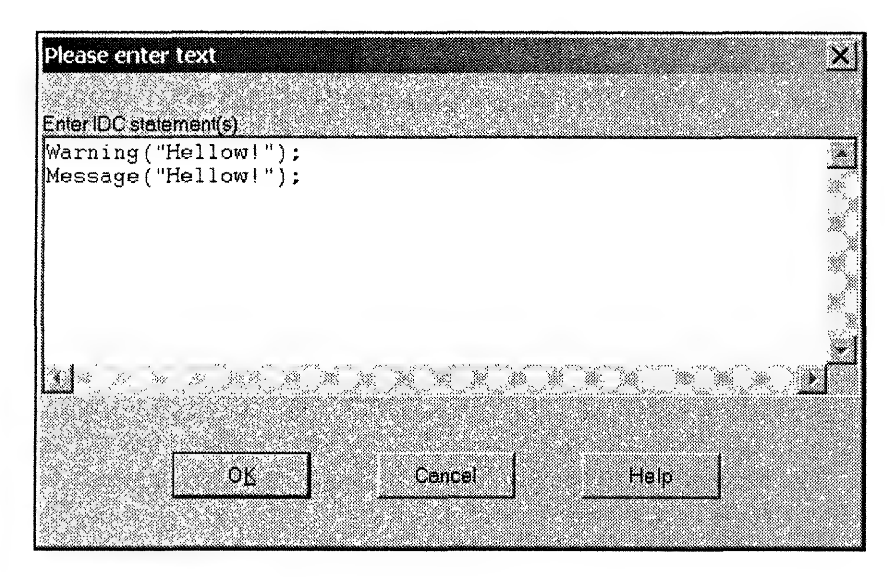


Рис. 5.8. Командное окно для выполнения последовательности инструкций языка IDC

Более основательный подход заключается в создании файла с расширением idc, который содержит текст на языке IDC. Для загрузки программы используется команда меню **File | Idc file**. При этом программа компилируется

и сразу выполняется. Кроме этого, в главном окне IDA Pro появляется панель (рис. 5.9) с кнопками для запуска и редактирования программы.

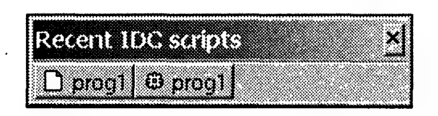


Рис. 5.9. Панель запуска и редактирования программы

Структура программы и синтаксис языка IDC

Рассмотрим структуру программ и синтаксис языка IDC.

Функции

Как и в языке С, в языке IDC программа состоит из функций, а выполнение программы начинается с выполнения функции `main`. Структура функции имеет следующий вид:

```
static func(arg1, arg2, ...)
{
...
}
```

Все функции должны объявляться как `static`. При задании аргументов не нужно указывать их тип. Последнее связано с тем, что в языке имеются только два типа переменных: строковые и числовые, принадлежность к которым легко определить по первой операции присвоения. Все же преобразования типов осуществляются автоматически.

Переменные

Все переменные являются локальными и объявляются при помощи ключевого слова `auto`. Существуют два типа переменных: числовые и строковые. Строковая переменная может содержать до 255 символов. Числовые переменные делятся на два типа: 32-битные целые (со знаком) числа и числа с плавающей точкой. Тип переменной определяется транслятором при ее первом присвоении.

Особо следует обратить внимание на преобразование типов. Рассмотрим различные ситуации.

- Преобразование строкового типа в числовой целый тип. Если левая часть строки является десятичным числом, то результат будет равен этому числу, в противном случае результат равен нулю.

Пример:

```
auto  a,b,c,d;
c="w"; d="q";
a="451";
b="123qwert234";
c=a; d=b;
Message("%d:%d\n",c,d);
```

Будет напечатано `451:123`.

Замечание

Функция `Message` встроенного языка IDC осуществляет вывод информации в окно сообщений (или *консоль сообщений*). Это окно IDA Pro открывает при запуске. Туда, в частности, выводятся сообщения о загрузке и анализе исполняемого кода. Функция `Message` является аналогом стандартной функции `printf` языка C.

- Преобразование числового целого типа в строковый тип. Преобразование довольно необычно для тех, кто привык, что такое преобразование просто сводится к замене числа на строку без изменения значения (2345 -> `"2345"`). Суть преобразования заключается в следующем: каждый байт числа справа налево преобразуется в символ в соответствующей кодировке, но размещается в строке слева направо. Например:

```
auto i1;
auto a;
a=0x4241;
i1="q";
i1=a;
Message("%s\n",i1);
```

В результате выполнения фрагмента будет выведена строка `"AB"`.

- Преобразование строкового типа в числовой тип с плавающей точкой. Данное преобразование осуществляется так же, как преобразование строкового типа в числовой целый тип.
- Преобразование числового типа с плавающей точкой в строковый тип. Здесь преобразование осуществляется по простой схеме: каждый разряд числа и десятичная точка преобразуются в соответствующий символ строки. При этом, однако, допускается некоторая потеря точности. Пример:

```
auto i1;
auto a;
a=" "; i1=3.5;
```

```
a=char(i1);
Message("%s\n",i1);
```

Результатом выполнения фрагмента будет строка:

```
3.50000000000000018318681
```

Преобразование типов может возникнуть не только при присвоении, когда тип, стоящий справа от знака равенства, преобразуется к типу, стоящему слева от знака равенства. Преобразование типов происходит также:

- если в арифметическом выражении имеется хотя бы один числовой тип с плавающей точкой, то все переменные выражения преобразуются к этому типу, так что действия в выражении осуществляются уже над вещественными числами;
- если над переменной производятся битовые операции, то переменная рассматривается как переменная целого числового типа.

Над переменными числового типа можно выполнять следующие операции: присвоение, сравнение, сложение, вычитание, умножение, деление. Кроме этого, над целыми переменными можно выполнять битовые операции:

- >>, << — циклические сдвиги;
- & — битовое "И";
- | — битовое "ИЛИ";
- ~ — битовое "НЕ";
- ^ — битовое "исключающее ИЛИ".

Над целыми переменными можно также выполнять операции инкремента (++) и декремента (--).

Над строковыми переменными можно осуществлять следующие операции:

- = — присвоение;
- == — сравнение;
- + — конкатенация.

Основные конструкции

Язык IDC поддерживает основные конструкции языка С, изменяющие ход выполнения программы:

- условные конструкции `if`, `else`;
- циклы `while`, `do`, `break`, `continue`;
- цикл с параметром (счетчиком) `for`;
- оператор возврата из функции `return`.

В языке отсутствуют такие операторы языка С, как `goto` и `switch`.

Директивы

Язык IDC поддерживает следующие препроцессорные директивы, используемые в языке С:

- `#define`;
- `#undef`;
- `#include`;
- `#error`;
- `#ifdef`, `#ifndef`, `#else`, `#endif`.

Управление строками

Язык IDC поддерживает минимальный набор работы со строковыми переменными. В отличие от языка С, строка здесь является не последовательностью символов, а некоторым закрытым элементом (или объектом) неопределенной структуры, для которого существуют операция конкатенации (слияния[6]) и несколько простых функций.

Для операции конкатенации используется обычный знак +. Например:

```
auto  s1,s2,s3;
s1="Hello,";
s2="world!";
s3=s1+" "+s2;
Message("%s\n",s3);
```

Результатом выполнения фрагмента будет вывод в консоль сообщений строки "Hello, world!"

Перечислю основные функции для работы со строками.

- `strlen` — возвращает длину строки. Единственным параметром функции является строковая переменная или константа.
- `strstr` — осуществляет поиск подстроки в строке. Первым аргументом функции является строка, где будет осуществляться поиск, вторым аргументом — подстрока для поиска. Функция возвращает номер символа, с которого начинается найденная подстрока. Нумерацию символов в строке принято отсчитывать от нуля. Если подстрока не найдена, то функция возвращает −1.
- `substr` — функция выделяет (и возвращает) подстроку в строке. Первым параметром функции является строка, над которой будет осуществляться операция. Второй и третий параметры функции — это начальный и ко-

---

[6] От англ. *to concatenate* — сцеплять, связывать, объединять.

нечный номера выделяемого фрагмента. Номера символов отсчитываются от 0. Функция возвращает выделенный фрагмент строки.

- `ltoa` — функция преобразует целое число в строку. Первый аргумент функции — числовая переменна или константа, второй аргумент — система счисления, в которой будет представлено число. Функция возвращает строку, представляющую число в заданной системе счисления. В случае ошибки возвращается пустая строка.
- `atoll` — функция осуществляет преобразование строки в целое число. Единственным аргументом функции является строка. В случае ошибки возвращается 0.

5.2.2. Встроенные функции и примеры программирования на языке IDC

Данный раздел не является справочником встроенных функций языка IDC, тем более, что в справочной системе IDA Pro имеется перечень этих функций и, кроме того, есть хорошая книга Криса Касперски "Образ мышления — дизассемблер IDA"[7], где практически все функции описаны очень подробно. Я попытаюсь сделать небольшой обзор функций, наиболее важных для анализа текста программ, и приведу несколько примеров их использования, отталкиваясь от которых можно писать свои небольшие программы анализа.

Кроме справочной системы IDA Pro, информацию о встроенных функциях можно почерпнуть также в файле idc.idc (подкаталог IDC), где хранятся определения констант и прототипы функций вместе с краткими к ним комментариями. Этот файл предназначен для подключения к программам на языке IDC, что мы и будем делать при помощи директивы `#include`. Кроме этого, в том же каталоге имеются несколько примеров программ на языке IDC, которые могут оказаться довольно полезны.

Доступ к виртуальной памяти

Напоминаю, что дизассемблер IDA Pro перед тем, как анализировать исполняемый модуль, создает виртуальную память, куда потом и загружает этот модуль. Обращаясь к конкретным ячейкам этой виртуальной памяти, мы тем самым обращаемся к загруженному туда программному коду, который к тому же еще предварительно проанализирован IDA Pro.

---

[7] Крис Касперски. Образ мышления — дизассемблер IDA. — М.: СОЛОН-Р, 2004.

Передвижение по памяти

Рассмотрим следующую программу на языке IDC (листинг 5.8).

Листинг 5.8

```
#include <idc.idc>
static main()
{
  auto ad;
  ad=0x401020;
  while(ad<=0x401041)
  {
    Message("%x\n",ad);
    ad=NextAddr(ad);
  };
}
```

Конечно, читателю, привыкшему ориентироваться в программах на языке С, не составляет труда осмыслить эту программу. С функцией `Message` мы уже хорошо знакомы, и остается только функция `NextAddr`. Понять ее смысл можно просто из названия. Функция возвращает следующий по отношению к значению своего аргумента линейный адрес. В случае если адрес не существует, то функция возвращает −1. Для этого значения в файле idc.idc придумана константа `BADADDR`.

Результатом выполнения программы будет столбик адресов с адреса `0x401041` по адрес `0x401041` включительно. Разумеется, к тому же результату мы придем, если просто на каждом витке цикла будем добавлять к переменной `ad` единицу. Имеется также функция `PrevAddr`, полностью аналогичная функции `NextAddr`, но возвращающая предыдущий адрес.

Наконец, существует еще одна весьма полезная функция, с помощью которой можно выполнять поиск (передвигаться) последовательности байтов в дизассемблированном тексте. Это функция `FindBinary`. Первым ее аргументом является адрес, начиная с которого следует осуществлять поиск. Второй аргумент — флаг режима поиска. Нулевой бит флага определяет направление поиска (0 — прямой поиск, 1 — поиск в обратном направлении), первый бит указывает на чувствительность к регистрам символов (0 — не различать регистры, 1 — различать регистры). Третий аргумент функции — последовательность кодов разыскиваемых байтов. Байты пишутся через пробел и заключаются в кавычки. Используется текущая система счисления. Функция возвращает адрес, с которого начинается искомая строка. В случае

если строка не будет найдена, функция возвращает –1. Пример вызова функции:

```
ad=FindBinary(0x404020,0,"34 AF 56 30")
```

Чтение и запись

Как я уже упоминал, функция `NextAddr` (и `PrevAddr`) может возвращать –1, если следующий виртуальный адрес не существует. Это значит, что следующий адрес либо недоступен, либо неинициализирован. А как быть в том случае, если команда просто обращается по какому-то адресу? Как заранее узнать, существует или нет данный адрес? Для этого предназначена функция `GetFlags`, единственным элементом которой является виртуальный адрес. Функция возвращает флаги этого адреса (атрибут). Нужные нам флаги проверяются при помощи константы `FF_IVL` (листинг 5.9), значение которой определено в файле idc.idc.

Для чтения из виртуальной памяти в языке IDC предусмотрены три функции: `Byte`, `Word` и `Dword`. Аргументами всех трех функций является виртуальный адрес. Соответственно функции возвращают байт, слово и двойное слово.

В листинге 5.9 представлена программа, которая читает блок виртуальной памяти и выводит его в окно сообщений.

Листинг 5.9

```
#include <idc.idc>

static main()
{
auto ad,i;
  for(ad=0x401020; ad<=0x401041; ad++)
  {
    Message("%x........",ad);
    if(GetFlags(ad) & FF_IVL)
    {
//печатаем значение прочитанного байта
      i=Byte(ad);
      if(i>31)
      Message("%x...%c\n",i,i);
      else
      Message("%x...\n",i);
    }else
```

```
    {
//значение байта не определено
      Message("Error!\n");
    }
  }
}
```

Результат выполнения программы из листинга 5.9 можно увидеть в листинге 5.10.

Листинг 5.10

```
401020........8b...‹
401021........44...D
401022........24...$
401023........4...
401024........6a...j
401025........0...
401026........68...h
401027........0...
401028........10...
401029........40...@
40102a........0...
40102b........6a...j
40102c........0...
40102d........68...h
40102e........ec...м
40102f........50...P
401030........40...@
401031........0...
401032........50...P
401033........ff...я
401034........15...
401035........c8...И
401036........50...P
401037........40...@
401038........0...
401039........6a...j
```

```
40103a........0...
40103b........ff...я
40103c........15...
40103d........0...
40103e........50...P
40103f........40...@
401040........0...
401041........cc...M
```

Для записи в виртуальную память используются три функции: `PatchByte`, `PatchWord`, `PatchDword`. Первым аргументом функций является адрес виртуальной памяти, вторым аргументом — записываемая в память величина. В листинге 5.11 представлена простая программа (не требующая комментария), анализирующая заданный блок памяти и изменяющая значения некоторых байтов.

Листинг 5.11

```
#include <idc.idc>

static main()
{
  auto ad,i,j;
  j=0x91;
  for(ad=0x401020; ad<=0x401041; ad++)
  {
    if(GetFlags(ad) & FF_IVL)
    {
      i=Byte(ad);
      if(i==0x50)PatchByte(ad,j);
    }
  }
}
```

Структура строки листинга

В строке листинга IDA Pro можно увидеть следующие элементы: инструкцию процессора или данное, комментарий, метку или перекрестную ссылку. Это уже не обезличенные данные, с которыми мы имели дело в предыдущем разделе. С другой стороны, со строкой листинга связаны определенные ячейки виртуальной памяти, в которых и хранятся коды инструкций или

данные. Рассмотрим, какие же функции можно использовать для анализа строки листинга дизассемблера.

Я намеренно говорю "строка листинга", дабы объединить воедино разные элементы. Различные, прежде всего, по тому, где они хранятся. Если инструкции и данные располагаются в виртуальной памяти (файл с расширением id1), то остальные перечисленные выше элементы располагаются в специальных виртуальных массивах, которые хранятся в файле с расширением id0. Однако для нас все это элементы строки, и поэтому я объединяю их в одном разделе.

Выделение инструкций

В листинге 5.12 представлена программа, выводящая на консоль сообщений ассемблерный текст в заданном промежутке адресов.

Листинг 5.12

```
#include <idc.idc>
static main()
{
  auto ad,i,j;
  ad=0x401000;
  while(ad<=0x401042)
  {
//операнды представить в шестнадцатеричном виде
    OpHex(ad,-1);
//вывести адрес инструкции
    Message("%10x ",ad);
//получить типы операндов
    i=GetOpType(ad,0);
    j=GetOpType(ad,1);
//вывести имя инструкции
    Message("%s ",GetMnem(ad));
    if(i>0)
    {
//вывести первый операнд (если он есть)
      Message("%s",GetOpnd(ad,0));
      if(j>0)
      {
//вывести второй операнд (если он есть)
        Message(",%s \n",GetOpnd(ad,1));
```

```
        }else
          Message("\n");
      }else
        Message("\n");
//перейти к адресу следующей инструкции
      ad=NextHead(ad,BADADDR);
   }
}
```

Комментарий к программе из листинга 5.12

- Главной в листинге является, конечно, функция NextHead. Первым аргументом этой функции выступает некоторый виртуальный адрес. Второй аргумент — адрес, ограничивающий значение возвращаемого адреса. Я использую BADADDR, который в данном случае трактуется как положительное целое число, т. е. FFFFFFFFH (а не −1). Функция возвращает адрес первого байта следующей инструкции или данного. Есть и аналогичная функция, но возвращающая адрес предыдущей инструкции или данного — PrevHead.
- Функция GetMnem возвращает по указанному адресу имя инструкции (строку), которая с этого адреса начинается. Аргументом функции является адрес первого байта инструкции.
- Функция GetOpnd возвращает операнд инструкции в виде строкового значения. Функция имеет два аргумента: адрес инструкции и номер (за минусом 1) операнда в инструкции, отсчитываемый слева направо.
- Для форматирования выводимой таблицы мне пришлось также использовать функцию GetOpType. Функция возвращает тип операнда инструкции процессора. Первым аргументом функции является адрес инструкции, вторым — номер (за минусом 1) операнда в инструкции, отсчитываемый слева направо. Мы воспользовались просто тем фактом, что если операнд присутствует, то значение, которое возвращает функция, должно быть больше нуля.
- Наконец, при помощи функции OpHex я задаю для каждой инструкции шестнадцатеричный формат вывода числовых операндов (если операнд — это число). Второй аргумент функции задает номер операнда. Значение −1 в программе означает, что функция должна распространять свое действие на все операнды инструкции.

В листинге 5.13 представлен результат выполнения программы из листинга 5.12.

Листинг 5.13

```
401000 push ebp
401001 mov  ebp,esp
401003 sub  esp,0Ch
401006 mov  dword ptr [ebp-4],0Ah
40100d mov  dword ptr [ebp-8],0Bh
401014 mov  dword ptr [ebp-0Ch],0Ch
40101b mov  eax,[ebp-0Ch]
40101e push eax
40101f mov  ecx,[ebp-8]
401022 push ecx
401023 mov  edx,[ebp-4]
401026 push edx
401027 call sub_401050
40102c add  esp,0Ch
40102f push eax
401030 push 4060D0h
401035 call _printf
40103a add  esp,8
40103d xor  eax,eax
40103f mov  esp,ebp
401041 pop  ebp
401042 retn
```

Выделение данных

Рассмотрим теперь, как разбирать имеющиеся в дизассемблерном листинге данные. Каждое данное занимает, по крайней мере, один байт. Какое данное начинается с указанного адреса, можно определить по битам атрибута байта, который находится по этому адресу. Типы данных и флаги для их определения из файла idc.idc представлены в листинге 5.14.

Листинг 5.14

```
#define FF_BYTE      0x00000000L   // byte
#define FF_WORD      0x10000000L   // word
#define FF_DWRD      0x20000000L   // dword
#define FF_QWRD      0x30000000L   // qword
#define FF_TBYT      0x40000000L   // tbyte
```

```
#define FF_ASCI      0x50000000L   // ASCII ?
#define FF_STRU      0x60000000L   // Struct ?
#define FF_OWRD      0x70000000L   // octaword (16 bytes)
#define FF_FLOAT     0x80000000L   // float
#define FF_DOUBLE    0x90000000L   // double
#define FF_PACKREAL  0xA0000000L   // packed decimal real
#define FF_ALIGN     0xB0000000L   // alignment directive
```

В листинге 5.15 представлена программа, которая выводит на консоль сообщений адреса данных, их длину и тип.

Листинг 5.15

```
#include <idc.idc>
static main()
{
  auto ad,i,j;
  ad=0x4055d6;
  while(ad<=0x405Aff)
  {
    ad=NextHead(ad,BADADDR);
//вывести адрес инструкции
    Message("%10x ",ad);
//получить флаг
    i=GetFlags(ad);
//проверка — данные ли это
    if(((i & MS_CLS) == FF_DATA))
    {
      Message("Data: size - %d, type - ",ItemSize(ad),i);
      if((i & 0xF0000000) == FF_BYTE)
      {
        Message("byte\n");
        continue;
      }
      if((i & 0xF0000000) == FF_WORD)
      {
        Message("word\n");
        continue;
      }
```

```
if((i & 0xF0000000) == FF_DWRD)
{
  Message("dword\n");
  continue;
}
if((i & 0xF0000000) == FF_QWRD)
{
  Message("qword\n");
  continue;
}
if((i & 0xF0000000) == FF_TBYT)
{
  Message("tbyte\n");
  continue;
}
if((i & 0xF0000000) == FF_ASCI)
{
  Message("string ASCII\n");
  continue;
}
if((i & 0xF0000000) == FF_STRU)
{
  Message("structure\n");
  continue;
}
if((i & 0xF0000000) == FF_OWRD)
{
  Message("octaword\n");
  continue;
}
if((i & 0xF0000000) == FF_FLOAT)
{
  Message("float\n");
  continue;
}
if((i & 0xF0000000) == FF_DOUBLE)
{
  Message("double\n");
  continue;
```

```
      }
      if((i & 0xF0000000) ==  FF_PACKREAL)
      {
        Message("packed decimal real\n");
        continue;
      }
      if((i & 0xF0000000) == FF_ALIGN)
      {
        Message("align\n");
        continue;
      };
      Message("??\n");

    }
    else
      Message("?\n");
  }
}
```

Комментарий к листингу 5.15

- Мы видим, что в программе опять используется функция `NextHead`, которая является наиболее удобной функцией для передвижения по дизассемблированному тексту.
- Для определения типа данных мы учитываем флаги атрибута первого байта данных. Мы используем таблицу флагов из листинга 5.14, выделяя нужные биты командой `i & F0000000h`.
- Наконец, длину данных мы определяем при помощи функции `ItemSize`. Единственным аргументом данной функции является адрес первого байта данного.

Результат выполнения программы из листинга 5.15 представлен в листинге 5.16.

Листинг 5.16

```
4055d8 Data: size - 160, type - string ASCII
405678 Data: size - 25,  type - string ASCII
405698 Data: size - 177, type - string ASCII
405749 Data: size - 3,   type - align
40574c Data: size - 35,  type - string ASCII
```

```
40576f Data: size - 1,   type - align
405770 Data: size - 12,  type - structure
405a82 Data: size - 66,  type - string ASCII
405c84 Data: size - 2,   type — word
```

Другие элементы строки

Другие элементы строки — это комментарии (автоматически создаваемые и пользовательские), метки (программные метки и переменные) и перекрестные ссылки. Вы можете не только программно получать эти элементы, но и добавлять их в строку.

В листинге 5.17 представлен взятый из файла idc.idc и прокомментированный мной список возможных элементов и значений флагов первого байта инструкции или данного.

Листинг 5.17

```
#define FF_COMM 0x00000800L  // Has comment?
        // комментарий
#define FF_REF  0x00001000L  // has references?
        // перекрестная ссылка
#define FF_LINE 0x00002000L  // Has next or prev cmt lines ?
        // строка многострочного комментария
#define FF_NAME 0x00004000L  // Has user-defined name ?
        // пользовательская метка (имя)
#define FF_LABL 0x00008000L  // Has dummy name?
        // метка (имя)
#define FF_FLOW 0x00010000L  // Exec flow from prev instruction?
        // перекрестная метка с предыдущей инструкции
#define FF_VAR  0x00080000L  // Is byte variable ?
        // переменная (метка для данного)
```

В листинге 5.18 представлена программа, осуществляющая просмотр листинга, который генерирует IDA Pro, и поиск программных меток, которые затем выводятся на консоль сообщений. В строках, которые имеют метки, добавляется комментарий — строка "Метка".

Листинг 5.18

```
#include <idc.idc>
static main()
```

```
{
  auto ad,i,j;
  ad=0x401cfe;
  while(ad<=0x401d41)
  {
    ad=NextHead(ad,BADADDR);
//вывести адрес инструкции
    Message("%10x ",ad);
    i=GetFlags(ad);
    if(i & FF_LABL)
    {
      Message("%s \n",GetTrueName(ad));
      MakeComm(ad,"Метка!");
    } else Message("\n");
  }
}
```

Комментарий к листингу 5.18

Поиск строк с метками осуществляется переходом от одной строки к другой и проверкой соответствующего бита у первого байта элемента (инструкции или данного) с использованием константы FF_LABL. Для создания комментария в программе вызывается функция MakeComm. Первым аргументом функции является адрес строки, вторым аргументом — строка-комментарий.

Работа с функциями

Функция — это объект листинга, который может состоять из нескольких строк, содержащих инструкции. Функция имеет начальный и конечный адрес, а также другие свойства (листинг 5.19). Разбивка дизассемблированного кода на функции позволяет значительно улучшить читаемость листинга, а также понимание логики выполнения программы.

В листинге 5.19 представлен перечень флагов, определяющих свойства функций, взятый из файла idc.idc.

Листинг 5.19

```
#define FUNC_NORET  0x00000001L    // function doesn't return
      //не возвращает управление командой ret
#define FUNC_FAR  0x00000002L     // far function
      //функция возвращает управление инструкцией retf
```

```
#define FUNC_LIB  0x00000004L      // library function
      //библиотечная функция
#define FUNC_STATIC  0x00000008L   // static function
      //статическая функция
#define FUNC_FRAME  0x00000010L    // function uses frame pointer (BP)
      //функция использует регистр EBP для указателя
      //на локальные переменные и параметры
#define FUNC_USERFAR  0x00000020L  // user has specified far-ness
      //определена пользователем как далекая (far)
#define FUNC_HIDDEN  0x00000040L   // a hidden function
      //скрытая (свернутая) функция
#define FUNC_THUNK  0x00000080L    // thunk (jump) function
      //функция-переходник, содержащая только
      //инструкцию jmp
#define FUNC_BOTTOMBP  0x00000100L  // BP points to the bottom
                                    //of the stack frame
                                    //регистр EBP указывает на "дно"
                                    //стекового фрейма
```

В листинге 5.20 представлена программа, которая выводит в консоль сообщений имена функций в заданном интервале адресов и устанавливает комментарий для библиотечных функций.

Листинг 5.20

```
#include <idc.idc>
static main()
{
  auto ad,s,i;
  ad=0x401000;
  while(ad<=0x4030bc)
  {
    s=GetFunctionName(ad);
    Message("%s\n",s);
    i=GetFunctionFlags(ad);
    if(i & FUNC_LIB)
    {
      SetFunctionCmt(ad,"Это библиотечная функция",1);
    }
```

```
    ad=NextFunction(ad);
  }
}
```

Комментарий к листингу 5.20

- Для передвижения по функциям листинга, генерируемого IDA Pro, используются функции `NextFunction` и `PrevFunction`, единственным параметром которых является адрес функции. Функции возвращают: одна — адрес следующей функции (эту функцию мы используем в программе), другая — адрес предыдущей функции.
- Программа выводит на консоль имена всех встретившихся ей функций, которые возвращаются функцией языка IDC `GetFunctionName`. Аргументом функции может служить любой адрес, принадлежащий функции.
- Для получения флагов функций применяется функция `GetFunctionFlags`. Флаги перечислены в листинге 5.19.
- Программа устанавливает у всех библиотечных функций (которые признаются библиотечными IDA Pro) комментарий. Для этого используется функция `SetFunctionCmt`. Она имеет три аргумента: адрес функции, строковый комментарий, тип комментария. Для функций можно устанавливать два типа комментариев: постоянный (параметр 0) и повторяемый (параметр 1). Первый комментарий присутствует только перед определением функции, второй также дублируется (повторяется) во всех вызовах данной функции.

Элементы интерфейса с пользователем

Дизассемблер IDA Pro предоставляет минимальный набор функций для автоматизации управления вводом/выводом. Это вывод в консоль сообщений, которым мы уже многократно пользовались (функция `Message`), управление курсором в дизассемблерном листинге, управление несколькими видами диалоговых окон и некоторые другие функции.

В листинге 5.21 представлена простая программа, которая в заданном интервале адресов ищет последовательность из трех подряд идущих инструкций `PUSH` и перемещает курсор к этой группе команд. Для перемещения курсора используется команда `Jump`, аргументом которой является виртуальный адрес.

Листинг 5.21

```
#include <idc.idc>
static main()
```

```
{
  auto ad,s;
  ad=0x401000;
  while(ad<=0x4030bc)
  {
    if(GetMnem(ad)=="push"    &&
    GetMnem(NextHead(ad,BADADDR))=="push" &&
    GetMnem(NextHead(NextHead(ad,BADADDR),BADADDR))=="push"
    )
    {
//перевести курсор по найденному адресу
      Jump(ad);
//выход из цикла
      break;
    }
    ad=NextHead(ad,BADADDR);
  }
}
```

Другие возможности программного анализа листинга в IDA Pro

Не имея возможности осветить весь перечень особенностей языка IDC, точнее библиотеки функций, предоставляемых IDA Pro, остановимся на некоторых интересных моментах.

Структуры и перечисления

В дизассемблере IDA Pro имеются встроенные возможности, позволяющие автоматически распознавать и определять такие важные элементы, присущие языкам высокого уровня, как *структуры* и *перечисления*. В IDA Pro и структура, и перечисление обладают тремя характеристиками, позволяющими их идентифицировать:

- идентификатор структуры или перечисления;
- имя структуры или перечисления;
- индекс структуры или перечисления.

В листинге 5.22 представлена программа, выводящая на консоль сообщений список идентификаторов и имена всех структур, которые были обнаружены IDA Pro при анализе исполняемого кода.

Листинг 5.22

```
#include <idc.idc>
static main()
{
  auto n,i,s;
  n=0;
  while(n!=-1)
  {
    i=GetStrucId(n);
    s=GetStrucName(i);
    n=GetNextStrucIdx(n);
    Message("%x %s\n",i,s);
  }
}
```

Комментарий к листингу 5.22

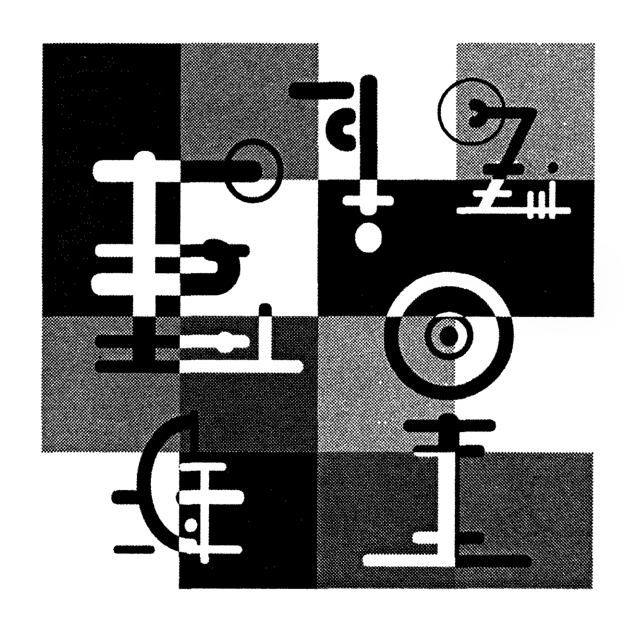
Функция `GetNextStrucIdx` возвращает следующий индекс структуры по заданному индексу, функция `GetStrucId` — идентификатор структуры по ее индексу, функция `GetStrucName` — имя структуры по ее индексу. Следует иметь в виду, что значения индексов структур и перечислений могут изменяться в процессе анализа (появление новых структур и уничтожение старых), идентификаторы же остаются неизменными.

Работа с файлами

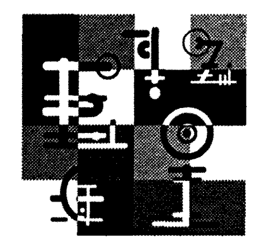
Встроенные функции позволяют работать с файлами. При помощи функции `GenerateFile` можно сгенерировать отчетный файл. Данная функция эквивалентна использованию пункта меню **File | Produce File**.

Дизассемблер IDA Pro поддерживает набор функций для управления файлами произвольной структуры. Этот набор функций в целом соответствует набору стандартных библиотечных файловых функций, определенных в заголовочных файлах stdio.h и io.h. Перечислю эти функции:

- `fopen` — открыть файл; функция возвращает дескриптор, который затем используется в других функциях;
- `flose` — закрыть файловый дескриптор;
- `filelength` — получить длину открытого при помощи `fopen` файла;
- `fgetc` — прочитать один символ из файла;
- `fputc` — записать один символ в файл;
- `ftell` — получить текущую позицию указателя;
- `fseek` — переместить указатель в заданную позицию в файле.



Приложения





Приложение 1

Представленная в листинге П1 программа проводит простейшее исследование PE-заголовка. Я не претендую на "чистоту" программирования. Единственной задачей, которую я перед собой ставил, — это демонстрация работы со структурами модуля PE.

Листинг П1

```
#include <windows.h>
#include <stdio.h>
HANDLE openf(char * );
DWORD getoffs(DWORD );
HANDLE hf;
DWORD n;
WORD m;
IMAGE_DOS_HEADER id;
IMAGE_NT_HEADERS iw;
IMAGE_SECTION_HEADER is;
IMAGE_SECTION_HEADER ais[100];
IMAGE_IMPORT_DESCRIPTOR im[1000];
IMAGE_THUNK_DATA it[1000];
IMAGE_IMPORT_BY_NAME in;
IMAGE_EXPORT_DIRECTORY ex;
IMAGE_RESOURCE_DIRECTORY rd1;
IMAGE_RESOURCE_DIRECTORY_ENTRY rde1[30];
IMAGE_RESOURCE_DIRECTORY rd2;
IMAGE_RESOURCE_DIRECTORY_ENTRY rde2[500];
IMAGE_COFF_SYMBOLS_HEADER ih;
IMAGE_DEBUG_DIRECTORY idd;
char * subs[]={"Unknown subsystem\n","Subsystem driver\n",
               "Subsystem GUI\n","Subsystem console\n",
               "Subsystem ?\n","Subsystem ?\n",
```

```
                 "Subsystem OS/2\n","Subsystem Posix\n"};
char   buf[300],buf1[300];;
DWORD im_n=0, it_n=0;
DWORD exn[5000];
WORD exo[5000];
DWORD exa[5000];
//главная функция
int main(int argc, char* argv[])
{
  int er=0,i;
  LARGE_INTEGER l;
//проверка наличия параметров
  if(argc<2){printf("No parameters!\n");er=1; goto _exit;};
//первый в списке - имя файла
  printf("File: %s\n",argv[1]);
  if((hf=openf(argv[1]))==INVALID_HANDLE_VALUE)
  {
    printf("No file!\n");
    er=2;
    goto _exit;};
//определить длину файла
  GetFileSizeEx(hf,&l);
//прочитать заголовок DOS
  if(!ReadFile(hf,&id,sizeof(id),&n,NULL))
  {
    printf("Read DOS_HEADER error 1!\n");
    er=3;
    goto _exit;};
  if(n<sizeof(id))
  {
    printf("Read DOS_HEADER error 2!\n");
    er=4;
    goto _exit;};
//проверить сигнатуру DOS ('MZ')
  if(id.e_magic!=IMAGE_DOS_SIGNATURE)
  {
    printf("No DOS signature!\n");
    er=5;
```

```
    goto _exit;}
  printf("DOS signature is OK!\n");
  if(id.e_lfanew>l.QuadPart)
  {
    printf("No NT signature!\n");
    er=6;
    goto _exit;};
//вначале передвинем указатель
  SetFilePointer(hf,id.e_lfanew,NULL,FILE_BEGIN);
//прочитать заголовок NT
  if(!ReadFile(hf,&iw,sizeof(iw),&n,NULL))
  {
    printf("Read NT_HEADER error 1!\n");
    er=7;
    goto _exit;};
  if(n<sizeof(iw))
  {
    printf("Read NT_HEADER error 2!\n");
    er=8;
    goto _exit;};
//проверить сигнатуру NT ('PE')
  if(iw.Signature!=IMAGE_NT_SIGNATURE)
  {
    printf("No NT signature!\n");
    er=9;
    goto _exit;}
  printf("NT signature is OK!\n");
//здесь поработаем над структурой
  printf("Number of sections %d\n",iw.FileHeader.NumberOfSections);
  printf("Size of optional header %d\n",
         iw.FileHeader.SizeOfOptionalHeader);
  if((iw.FileHeader.Characteristics&0x2000)!=0) printf("DLL-modul\n");
  else
  {
    if(((iw.FileHeader.Characteristics&0x1000)!=0)) printf("System
modul\n");
    printf("EXE-modul\n");
  };
  if(iw.FileHeader.Machine ==0x014c) printf("Processor Intel\n");
```

```
  else printf("Unknown processor\n");
//теперь опционный заголовок
  printf("Linker version %d.%d\n",
          iw.OptionalHeader.MajorLinkerVersion,
          iw.OptionalHeader.MinorLinkerVersion);
  printf("Size of code %d\n",iw.OptionalHeader.SizeOfCode);
  printf("Size of initialized data %d\n",
         iw.OptionalHeader.SizeOfInitializedData);
  printf("Size of uninitialized data %d\n",
         iw.OptionalHeader.SizeOfUninitializedData);
  printf("Address Of Entry Point (RVA) %xh\n",
         iw.OptionalHeader.AddressOfEntryPoint);
  printf("Address of code (RVA) %xh\n",iw.OptionalHeader.BaseOfCode);
  printf("Address of data (RVA) %xh\n",iw.OptionalHeader.BaseOfData);
  printf("Image Base %xh\n",iw.OptionalHeader.ImageBase);
  printf("Size Of Image %xh\n",iw.OptionalHeader.SizeOfImage);
  printf("Size of Headers %xh\n",iw.OptionalHeader.SizeOfHeaders);
  printf("Section Alignment %xh\n",iw.OptionalHeader.SectionAlignment);
  printf("File Alignment %xh\n",iw.OptionalHeader.FileAlignment);
  printf("Size Of Stack Reserve %d\n",
         iw.OptionalHeader.SizeOfStackReserve);
  printf("Size Of Stack Commit %d\n",
         iw.OptionalHeader.SizeOfStackCommit);
  printf("Size Of Heap Reserve %d\n",
         iw.OptionalHeader.SizeOfHeapReserve);
  printf("Size Of Heap Commit %d\n",iw.OptionalHeader.SizeOfHeapCommit);
  printf("%s",subs[iw.OptionalHeader.Subsystem]);
//список секций
//виртуальные адреса таблицы некоторых таблиц PE
  DWORD vi=iw.OptionalHeader.DataDirectory[1].VirtualAddress;
  DWORD ve=iw.OptionalHeader.DataDirectory[0].VirtualAddress;
  DWORD vr=iw.OptionalHeader.DataDirectory[2].VirtualAddress;
  DWORD vg=iw.OptionalHeader.DataDirectory[6].VirtualAddress;
//
  printf("Sections:\n");
  printf("    Name   sizev    sizef     adrf     adrv\n");
  printf("-----------------------------------------\n");
  int j=0;
```

```
  for(i=0; i<iw.FileHeader.NumberOfSections; i++)
  {
    if(!ReadFile(hf,&is,sizeof(is),&n,NULL))
    {
      printf("IMAGE_SECTION_HEADER error!\n");
      er=10;
      goto _exit;
    };
    printf("%8s %6xh  %6xh  %6xh  %6xh\n",
           is.Name,is.Misc.VirtualSize,is.SizeOfRawData,
           is.PointerToRawData,is.VirtualAddress);
    ais[i].VirtualAddress=is.VirtualAddress;
    ais[i].PointerToRawData=is.PointerToRawData;
  };
  printf("-------------------------------------------\n");
  printf("\n");
//таблица импорта
  if(!vi)
  {
    printf("No import!\n");
  }else
  {
    printf("Import section offset %xh virtual %xh\n",getoffs(vi),vi);
//вначале передвинем указатель
    SetFilePointer(hf,getoffs(vi),NULL,FILE_BEGIN);
    while(TRUE)
    {
      if(!ReadFile(hf,&im[im_n],sizeof(im[im_n]),&n,NULL))
      {
        printf("IMAGE_IMPORT_DESCRIPTOR error!\n");
        er=11;
        goto _exit;
      };
      if(im[im_n].Characteristics==0&&im[im_n].Name==0)break;
      im_n++;
    };
//библиотеки
    printf("Import objects:\n");
```

```
    for(i=0; i<(int)im_n; i++)
    {
    //вначале библиотеки DLL
    SetFilePointer(hf,getoffs(im[i].Name),NULL,FILE_BEGIN);
      ReadFile(hf,buf,100,&n,NULL);
      printf("%s\n",buf);
    //теперь названия функций
      if(im[i].OriginalFirstThunk!=0)
        SetFilePointer(hf,getoffs(im[i].OriginalFirstThunk),
                       NULL,FILE_BEGIN);
      else
        SetFilePointer(hf,getoffs(im[i].FirstThunk),NULL,FILE_BEGIN);
      it_n=0;
      printf("Offset of AdresImpArray %xh RVA of AdresImpArray %xh\n",
             getoffs(im[i].FirstThunk),im[i].FirstThunk);
      while(TRUE)
      {
      ReadFile(hf,&it[it_n],sizeof(it[it_n]),&n,NULL);
        if(it[it_n].u1.AddressOfData==0)break;
        it_n++;
      };
      for(j=0; j<(int)it_n; j++)
      {
        if((it[j].u1.AddressOfData&IMAGE_ORDINAL_FLAG32)==0)
        {
          SetFilePointer(hf,getoffs(it[j].u1.ForwarderString+2),
                         NULL,FILE_BEGIN);
          ReadFile(hf,buf,100,&n,NULL);
          printf("     %s %xh %xh\n",buf,
                 getoffs(it[j].u1.ForwarderString+2),
                 it[j].u1.ForwarderString+2);
        } else printf("   Ordinal %d\n",
                      it[j].u1.AddressOfData&0x0000ffff);
      };
    };
  };
//таблица экспорта
  printf("\n");
```

```
  if(!ve)
  {
    printf("No export!\n");
  }else
  {
    printf("Export section offset %xh virtual %xh\n",getoffs(ve),ve);
    SetFilePointer(hf,getoffs(ve),NULL,FILE_BEGIN);
    if(!ReadFile(hf,&ex,sizeof(ex),&n,NULL))
    {
      printf("IMAGE_EXPORT_DIRECTORY error!\n");
      er=12;
      goto _exit;
    };
    SetFilePointer(hf,getoffs(ex.Name),NULL,FILE_BEGIN);
    ReadFile(hf,buf,100,&n,NULL);
    printf("Export modul: %s\n",buf);
    printf("Number of functions: %d\n",ex.NumberOfFunctions);
    printf("Number of names: %d\n",ex.NumberOfNames);
    printf("Ordinal base %d\n",ex.Base);
//массив указателей на имена экспортируемых функций
  SetFilePointer(hf,getoffs(ex.AddressOfNames),NULL,FILE_BEGIN);
    for(i=0; i<ex.NumberOfNames; i++)
      ReadFile(hf,&exn[i],4,&n,NULL);
//массив указателей на ординалы экспортируемых функций
    SetFilePointer(hf,getoffs(ex.AddressOfNameOrdinals),NULL,FILE_BEGIN);
    for(i=0; i<ex.NumberOfNames; i++)
      ReadFile(hf,&exo[i],2,&n,NULL);
//массив указателей на адреса экспортируемых функций
  SetFilePointer(hf,getoffs(ex.AddressOfFunctions),NULL,FILE_BEGIN);
    for(i=0; i<ex.NumberOfFunctions; i++)
      ReadFile(hf,&exa[i],4,&n,NULL);
    printf("\n");
    printf("Name of function                Ord     VAdr\n");
    printf("---------------------------------------------\n");
//вывод имен экспортируемых функций
    for(i=0; i<ex.NumberOfNames; i++)
    {
      SetFilePointer(hf,getoffs(exn[i]),NULL,FILE_BEGIN);
```

```
      ReadFile(hf,buf,300,&n,NULL);
      printf("%30s %4d %8xh\n",buf,exo[i]+ex.Base,exa[exo[i]]);
    };
    printf("--------------------------------------------\n");

  };
//работа с ресурсами
  printf("\n");
  if(!vr)
  {
    printf("No resource!\n");
  }else
  {
    DWORD offres=getoffs(vr);
    printf("Resource: offset %xh virtual %xh \n",offres,vr);
    SetFilePointer(hf,offres,NULL,FILE_BEGIN);
    ReadFile(hf,&rd1,sizeof(rd1),&n,NULL);
    //1-й уровень
    printf("Number of type %d \n",rd1.NumberOfIdEntries);
    //вначале пропустить rd.NumberOfNamedEntries записей
    for(i=0; i<rd1.NumberOfNamedEntries; i++)
      ReadFile(hf,&rde1[i],sizeof(rde1[i]),&n,NULL);
    //перечень типов ресурсов запомнить в массиве
    for(i=0; i<rd1.NumberOfIdEntries; i++)
      ReadFile(hf,&rde1[i],sizeof(rde1[i]),&n,NULL);
    //вывод типов ресурсов
    for(i=0; i<rd1.NumberOfIdEntries; i++)
      printf("Type identify: %d\n",rde1[i].Name);
    //2-й уровень
    printf("List of resource:\n");
    for(i=0; i<rd1.NumberOfIdEntries; i++)
    {
      SetFilePointer(hf,
        (rde1[i].OffsetToData & 0x7fffffff)+offres,NULL,FILE_BEGIN);
      ReadFile(hf,&rd2,sizeof(rd2),&n,NULL);
      printf("*Type of resource: %d\n",rde1[i].Id);
      for(j=0; j<rd2.NumberOfNamedEntries+rd2.NumberOfIdEntries; j++)
```

```
        ReadFile(hf,&rde2[j],sizeof(rde2[j]),&n,NULL);
      for(j=0; j<rd2.NumberOfNamedEntries+rd2.NumberOfIdEntries; j++)
      {
        if(!(rde2[j].Name & 0x80000000))
        {
          printf(" -Resource identify %d\n",rde2[j].Name);
        }
        else
        {
          SetFilePointer(hf,(rde2[j].Name & 0x7fffffff)+offres,
                        NULL,FILE_BEGIN);
          ReadFile(hf,&m,2,&n,NULL);
          ReadFile(hf,buf,2*m,&n,NULL);
          //перекодировка из Unicode
          WideCharToMultiByte(CP_UTF7,0,(LPCWSTR)(buf),m,buf1,300,
                             NULL,NULL);
          printf(" -Name of resource: %s\n",buf1);
        }
      };
    };

  };
//проверим отладочную информацию
  printf("\n");
  if(!vg)
  {
    printf("No debug table!\n");
  }else
  {
    DWORD offdbg=getoffs(vg);
    printf("Debug table: offset %xh virtual %xh \n",offdbg,vg);
    SetFilePointer(hf,offdbg,NULL,FILE_BEGIN);
    ReadFile(hf,&idd,sizeof(idd),&n,NULL);
    printf("Type of debug information: %d\n",idd.Type);
    //для COFF-информации
    if(idd.Type==1)
    {
    SetFilePointer(hf,idd.PointerToRawData,NULL,FILE_BEGIN);
```

```
      ReadFile(hf,&ih,sizeof(ih),&n,NULL);
      printf("RVA of first line number: %xh\n",
             idd.PointerToRawData+ih.LvaToFirstLinenumber);
    }
  }
  if(!iw.FileHeader.PointerToSymbolTable )
  {
    printf("No symbol table!\n");
  }else
  {
    DWORD offsym=getoffs(iw.FileHeader.PointerToSymbolTable);
    printf("Symbol table: offset %xh virtual %xh \n",
           offsym,iw.FileHeader.PointerToSymbolTable);
  };

//закрыть дескриптор файла
_exit:
  CloseHandle(hf);
  return er;
};
//функция открывает файл для чтения
HANDLE openf(char * nf)
{
  return CreateFile(nf,
    GENERIC_READ,
    FILE_SHARE_WRITE | FILE_SHARE_READ,
    NULL,
    OPEN_EXISTING,
    NULL,
    NULL);
};
//определение смещения в файле PE по относительному виртуальному адресу
DWORD getoffs(DWORD vsm)
{
  DWORD fi=0;
  if(vsm<ais[0].VirtualAddress)return fi;
  for(int i=0; i<iw.FileHeader.NumberOfSections; i++)
```

```
  {
    if(vsm<ais[i].VirtualAddress&&i>0){
    fi=ais[i-1].PointerToRawData+(vsm-ais[i-1].VirtualAddress);
      break;};
  };
  if(i==iw.FileHeader.NumberOfSections)
    fi=ais[i-1].PointerToRawData+(vsm-ais[i-1].VirtualAddress);
  return fi;
};
```

В листинге П2 представлен пример работы программы с одним из загружаемых модулей.

Листинг П2

```
File: primer42.exe
DOS signature is OK!
NT signature is OK!
Number of sections 4
Size of optional header 224
EXE-modul
Processor Intel
Linker version 5.12
Size of code 1024
Size of initialized data 2048
Size of uninitialized data 0
Address Of Entry Point (RVA) 1000h
Address of code (RVA) 1000h
Address of data (RVA) 2000h
Image Base 400000h
Size Of Image 5000h
Size of Headers 400h
Section Alignment 1000h
File Alignment 200h
Size Of Stack Reserve 1048576
Size Of Stack Commit 4096
Size Of Heap Reserve 1048576
```

```
Size Of Heap Commit 4096
Subsystem GUI
Sections:
    Name   sizev    sizef     adrf      adrv
-------------------------------------------
   .text    214h     400h     400h     1000h
  .rdata    23eh     400h     800h     2000h
   .data     91h     200h     c00h     3000h
   .rsrc    150h     200h     e00h     4000h
-------------------------------------------

Import section offset 8a4h virtual 20a4h
Import objects:
user32.dll
Offset of AdresImpArray 80ch RVA of AdresImpArray 200ch
     CreateWindowExA 94eh 214eh
     DefWindowProcA 960h 2160h
     DispatchMessageA 972h 2172h
     GetMessageA 986h 2186h
     LoadCursorA 994h 2194h
     MessageBoxA 940h 2140h
     PostQuitMessage 9aeh 21aeh
     RegisterClassA 9c0h 21c0h
     ShowWindow 9d2h 21d2h
     TranslateMessage 9e0h 21e0h
     UpdateWindow 9f4h 21f4h
     LoadMenuA 934h 2134h
     LoadIconA 9a2h 21a2h
     SetMenu 92ah 212ah
kernel32.dll
Offset of AdresImpArray 800h RVA of AdresImpArray 2000h
     ExitProcess a10h 2210h
     GetModuleHandleA a1eh 221eh

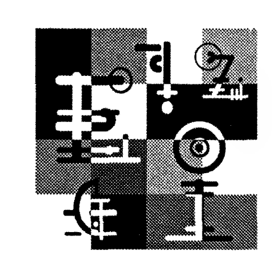
No export!

Resource: offset e00h virtual 4000h
```

```
Number of type 1
Type identify: 4
List of resource:
*Type of resource: 4
 -Name of resource: MENUP

Debug table: offset 850h virtual 2050h
Type of debug information: 1
RVA of first line number: 1000h
Symbol table: offset 420h virtual 1020h
```

Приложение 2



Описание компакт-диска

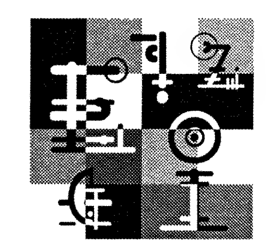
Содержимое диска разбито на каталоги. Каждый каталог соответствует своей главе или приложению. В свою очередь каждый каталог состоит из подкаталогов, где хранятся листинги, приведенные в тексте книги. Названия каталогов и подкаталогов соответствуют номерам глав и номерам листингов. В подкаталогах pictures хранятся схемы, используемые в книге.

Листинги, представляющие собой программы, хранятся в виде полных проектов, так чтобы их можно было сразу загрузить и откомпилировать в соответствующей среде программирования (например, Visual C++ или Borland C++). Для всех программ в каталоге содержится и уже готовый исполняемый модуль. Это относится и к программам на языке ассемблера, для трансляции которых использовался транслятор MASM32. Другие листинги представляют собой файлы в формате ASCII.

Каждый подкаталог содержит файл read.me в формате ASCII с кратким пояснением к листингу.

Литература





Книги автора по программированию в операционной системе Windows:

1. Пирогов В. Ю. Ассемблер для Windows. 3-е изд. — СПб.: БХВ-Петербург, 2005.
2. Пирогов В. Ю. Ассемблер на примерах. — СПб.: БХВ-Петербург, 2005.
3. Пирогов В. Ю. Ассемблер учебный курс. 2-е изд. — СПб.: БХВ-Петербург, 2003.
4. Пирогов В. Ю. Программирование на Visual C++ .NET. — СПб.: БХВ-Петербург, 2003.

Предметный указатель

K

M

O

P

R

S

T

U

V

W

А

Б

В

Г

Д

И

К

М

Н

О

П

Ф

Ц

Ч

Э

Я



www.bhv.ru

Книги издательства "БХВ-Петербург" в продаже:



www.bhv.ru

Книги издательства "БХВ-Петербург" в продаже:

Внесерийные книги

Mandrakesoft. Установка и использование Mandrakelinux 10.0 (+CD-ROM)	144 с.
MandrakeSoft Mandrakelinux. Полное руководство пользователя	528 с.
Mandriva Linux: полное руководство пользователя	544 с.
Актершев С. Задачи на максимум и минимум	199 с.
Андрианов В., Соколов А. Автомобильные охранные системы. Справочное пособие	272 с.
Бернс С. Фотомагия Photoshop. Трюки и эффекты. Полноцветное издание	448 с.
Богданов-Катьков Н. Струйные принтеры для дома и офиса	224 с.
Боков В. Физика магнетиков. Учебное пособие для вузов	129 с.
Боресков А. Разработка и отладка шейдеров	496 с.
Брайант Р., О'Халарон Д. Компьютерные системы: архитектура и программирование	1104 с.
Брэй Б. Микропроцессоры Intel: 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, Pentium Pro Processor, Pentium II, Pentium III, Pentium 4.Архитектура, программирование и интерфейсы. Шестое издание	1328 с.
Бурлаков М. В. Путеводитель по Adobe Photoshop CS 2	688 с.
Бутиков Е. Оптика: Учебное пособие для студентов физических специальностей вузов, 2-е изд.	480 с.
Быков А. и др. ADEM CAD/CAM/TDM. Черчение, модернизация, механообработка (+CD-ROM)	320 с.
Гасфилд Д. Строки, деревья и последовательности в алгоритмах	654 с.
Гласс Г., Эйблс К. Unix для программистов и пользователей, 3-е изд.	848 с.
Гольдштейн Б. Стек протоколов ОКС7. Подсистема ISUP. Справочник	480 с.
Гольдштейн Б. Интерфейсы V5.1 и V5.2. Справочник	288 с.
Гольдштейн Б. Системы коммутации, 2-е изд.	318 с.
Гольдштейн Б. Call-центры и компьютерная телефония	372 с.
Гурова А. Герои меча и магии. По мотивам одноименной компьютерной игры	320 с.
Данилов П. The Bat! 3. Практическая работа	288 с.
Джазайери М, и др. Основы инженерии программного обеспечения. 2-е изд., пер. с англ.	832 с.

Дорот В., Новиков Ф. Толковый словарь современной компьютерной лексики, 3-е изд.	608 с.
Зыль С. QNX Momentics: основы применения (+CD-ROM)	256 с.
Зыль С. Операционная система реального времени QNX: от теории к практике, 2-е изд. (+CD-ROM)	192 с.
Иванов К. Сборник задач по элементарной математике для абитуриентов, 4-е изд.	352 с.
Калашников О. Ассамблер? Это просто! Учимся программировать	384 с.
Калиновский А. Ваша домашняя страничка в Интернете. Homepage, или просто "хомяк"	224 с.
Канторович Л., Акилов Г. Функциональный анализ, 4-е изд.	816 с.
Карпюк В. MS Windows XP Professional. Опыт сдачи сертификационного экзамена 70-270	528 с.
Кертен Р. Введение в QNX Neutrino 2. Руководство для разработчиков приложений реального времени	400 с.
Климов А. Занимательное программирование на Visual Basic .NET	528 с.
Климов А., Чеботарев И. Windows. Народные советы	400 с.
Корнеев В., Киселев А. Современные микропроцессоры, 3-е изд.	448 с.
Кохась К. Задачи Санкт-Петербургской олимпиады школьников по математике 2003 года	224 с.
Кохась К. Задачи Санкт-Петербургской олимпиады школьников по математике 2004 года	224 с.
Кулагин Б. 3ds max 8: Актуальное моделирование, визуализация и анимация	496 с.
Кулагин Б., Морозов Д. 3ds max 6 и character studio 4. Анимация персонажей	224 с.
Культин Н. Visual Basic. Освой на примерах (+CD-ROM)	288 с.
Культин Н. Visual Basic. Освой самостоятельно	480 с.
Макаров Б. и др. Избранные задачи по вещественному анализу, 2-е изд.	624 с.
Малыхина М. Базы данных: основы, проектирование, использование	512 с.
Мачник Э. Фотообман в Photoshop	272 с.
Михайлов С., Черков А., Цветков И. 1С:Бухгалтерия 7.7. Решение типичных проблем пользователя	272 с.
Морозова О. Построй свой супер-сайт за 21 день!	224 с.
Новиков Б., Домбровская Г. Настройка приложений баз данных	210 с.
Очков В. Mathcad 12 для студентов и инженеров	464 с.
Палмер М., Синклер Р. Проектирование и внедрение компьютерных сетей. Учебный курс, 2-е изд.	240 с.
Пахомов Б. C++ и Borland C++ Builder для начинающих	640 с.
Петров Ю. Новые главы теории управления и компьютерных вычислений	192 с.
Пирогов В. Ассемблер. Учебный курс. 2-е изд.	1056 с.
Пог Д. MS Windows XP Home Edition: недокументированные возможности	768 с.

Погорелов В. AutoCAD 2005 для начинающих	400 с.
Половко А. Интерполяция. Методы и компьютерные технологии их реализации	320 с.
Попов А. Администрирование Windows с помощью WMI и WMIC (+CD-ROM)	752 с.
Попов С. Аппаратные средства мультимедиа. Видеосистема PC	400 с.
Потопахин В. Turbo Pascal. Решение сложных задач	208 с.
Правин О. Правильный самоучитель работы на компьютере, 2-е изд.	496 с.
Прохоров А. Интернет: как это работает	280 с.
Роб П. Системы баз данных: проектирование, реализация и управление, 5-е изд.	299 с.
Роб П. Системы баз данных: проектирование, разработка и использование	1200 с.
Робачевский А. Операционная система UNIX	528 с.
Романовский И. Дискретный анализ, 3-е изд.	320 с.
Рыжиков Ю. Работа над диссертацией по техническим наукам	
Скляров Д. Искусство защиты и взлома информации	288 с.
Соколов А., Андрианов В. Альтернатива сотовой связи: транкинговые системы	448 с.
Соколов А., Степанюк О. Защита от компьютерного терроризма	126 с.
Соломенчук В. Knoppix — это Linux без проблем	336 с.
Соломенчук В. Железо ПК 2005	480 с.
Соломенчук В. Железо ПК 2006	448 с.
Столлингс В. Компьютерные сети, протоколы и технологии Интернета	650 с.
Суворов К., Черемных М. Справочник Delphi. Базовые классы	576 с.
Титтел Э., Чеппел Л. TCP/IP. Учебный курс (+CD-ROM)	976 с.
Феличи Д. Типографика: шрифт, верстка, дизайн	360 с.
Фленов М. Библия Delphi (+CD-ROM)	880 с.
Фленов М. Программирование в Delphi глазами хакера (+CD-ROM)	368 с.
Фленов М. Программирование на С++ глазами хакера (+CD-ROM)	336 с.
Фленов М. Linux глазами хакера	550 с
Фленов М. PHP глазами хакера	304 с.
Фленов М. Компьютер глазами хакера	336 с.
Фрей Д. AutoCAD и AutoCAD LT для начинающих	680 с.
Частиков А. Архитекторы компьютерного мира	384 с.
Чечельницкий А. Десятипальцевый набор на клавиатуре	48 с.
Шапорев Д. Visual FoxPro. Уроки программирования	480 с.
Щёлоков В. Новые правила приватизации земельных участков и строений. Справочник землепользователя	192 с.
Яцюк О. Основы графического дизайна на базе компьютерных технологий (+CD-ROM)	270 с.

АССЕМБЛЕР и ДИЗАССЕМБЛИРОВАНИЕ



Руководство по дизассемблированию и отладке Windows-приложений



Пирогов Владислав Юрьевич, кандидат физико-математических наук, профессор кафедры новых информационных технологий Шадринского государственного педагогического института, программист с 20-летним стажем. Автор нескольких книг по программированию, в том числе: «Ассемблер для Windows», «Ассемблер. Учебный курс» и др., а также популярного сайта «Ассемблер и не только» (**http://asm.shadrinsk.net**).

Знание ассемблера и основ дизассемблирования позволяет программисту, с одной стороны, эффективно строить защиту своих собственных программ, а с другой — писать более эффективный и оптимизированный программный код. Данное практическое руководство поможет понять механизмы функционирования исполняемых модулей в среде Windows, а также соответствие между структурами языка высокого уровня и машинного кода. Вы найдете практические примеры исследования исполняемого кода и узнаете основные принципы подобного исследования: идентификацию программных структур, поиск данных и др. Изучите инструменты, используемые для работы с исполняемым кодом. Освоите основы работы с программами SoftICE и IDA Pro, которые считаются в настоящее время наиболее мощными и «продвинутыми» в области дизассемблирования и отладки. Большое количество примеров, листингов программ и иллюстраций облегчат освоение материала книги.

КАТЕГОРИЯ:
АССЕМБЛЕР



Компакт-диск содержит тексты листингов, описанных в книге, а также учебные программы.





БХВ-ПЕТЕРБУРГ 194354, ул. Есенина, 5Б
E-mail: mail@bhv.ru Internet: www.bhv.ru
Тел./факс: (812) 591-6243